A Software-Defined Radio for the Masses, Part 1

This series describes a complete PC-based, software-defined radio that uses a sound card and an innovative detector circuit. Mathematics is minimized in the explanation. Come see how it's done.

By Gerald Youngblood, AC5OG

A certain convergence occurs when multiple technologies align in time to make possible those things that once were only dreamed. The explosive growth of the Internet starting in 1994 was one of those events. While the Internet had existed for many years in government and education prior to that, its popularity had never crossed over into the general populace because of its slow speed and arcane interface. The development of the Web browser, the rapidly accelerating power and availability of the PC, and the availability of inexpensive and increasingly speedy modems brought about the Internet convergence. Suddenly, it all came together so that the Internet and the worldwide Web joined the everyday lexicon of our society.

A similar convergence is occurring in radio communications through digital signal processing (DSP) software to perform most radio functions at performance levels previously considered unattainable. DSP has now been incorporated into much of the amateur radio gear on the market to deliver improved noise-reduction and digital-filtering performance. More recently, there has been a lot of discussion about the emergence of so-called software-defined radios (SDRs).

A software-defined radio is characterized by its flexibility: Simply modifying or replacing software programs can completely change its functionality. This allows easy upgrade to new modes and improved performance without the need to replace hardware. SDRs can also be easily modified to accommodate the operating needs of individual applications. There is a distinct difference between a radio that internally uses software for some of its functions and a radio that can be completely redefined in the field through modification of software. The latter is a software-defined radio.

This SDR convergence is occurring because of advances in software and silicon that allow digital processing of radio-frequency signals. Many of these designs incorporate mathematical functions into hardware to perform all of the digitization, frequency selection, and down-conversion to base-

8900 Marybank Dr
Austin, TX 78750
gerald@sixthmarket.com

band. Such systems can be quite complex and somewhat out of reach to most amateurs.

One problem has been that unless you are a math wizard and proficient in programming *C++* or assembly language, you are out of luck. Each can be somewhat daunting to the amateur as well as to many professionals. Two years ago, I set out to attack this challenge armed with a fascination for technology and a 25-year-old, virtually unused electrical engineering degree. I had studied most of the math in college and even some of the signal processing theory, but 25 years is a long time. I found that it really was a challenge to learn many of the disciplines required because much of the literature was written from a mathematician's perspective.

Now that I am beginning to grasp many of the concepts involved in software radios, I want to share with the Amateur Radio community what I have learned without using much more than simple mathematical concepts. Further, a software radio should have as little hardware as possible. If you have a PC with a sound card, you already have most of the required hardware. With as few as three integrated circuits you can be up and running with a Tayloe detector—an innovative, yet simple, direct-conversion receiver. With less than a dozen chips, you can build a transceiver that will outperform much of the commercial gear on the market.

Approach the Theory

In this article series, I have chosen to focus on practical implementation rather than on detailed theory. There are basic facts that must be understood to build a software radio. However, much like working with integrated circuits, you don't have to know how to create the IC in order to use it in a design. The convention I have chosen is to describe practical applications followed by references where appropriate for more detailed study. One of the easier to comprehend references I have found is *The Scientist and Engineer's Guide to Digital Signal Processing* by Steven W. Smith. It is free for download over the Internet at **www.DSPGuide.com**. I consider it required reading for those who want to dig deeper into implementation as well as theory. I will refer to it as the "DSP Guide" many times in this article series for further study.

So get out your four-function calculator (okay, maybe you need six or seven functions) and let's get started. But first, let's set forth the objectives of the complete SDR design:

- Keep the math simple
- Use a sound-card equipped PC to provide all signal-processing functions
- Program the user interface and all signal-processing algorithms in *Visual Basic* for easy development and maintenance
- Utilize the Intel Signal Processing Library for core DSP routines to minimize the technical knowledge requirement and development time, and to maximize performance
- Integrate a direct conversion (D-C) receiver for hardware design simplicity and wide dynamic range
- Incorporate direct digital synthesis (DDS) to allow flexible frequency control
- Include transmit capabilities using similar techniques as those used in the D-C receiver.

Analog and Digital Signals in the Time Domain

To understand DSP we first need to understand the relationship between digital signals and their analog counterparts. If we look at a 1-V (pk) sine wave on an analog oscilloscope, we see that the signal makes a perfectly smooth curve on the scope, no matter how fast the sweep frequency. In fact, if it were possible to build a scope with an infinitely fast horizontal sweep, it would still display a perfectly smooth curve (really a straight line at that point). As such, it is often called a *continuous-time signal* since it is continuous in time. In other words, there are an infinite number of different voltages along the curve, as can be seen on the analog oscilloscope trace.

On the other hand, if we were to measure the same sine wave with a digital voltmeter at a sampling rate of four times the frequency of the sine wave, starting at time equals zero, we would read: 0 V at 0°, 1 V at 90°, 0 V at 180° and –1 V at 270° over one complete cycle. The signal could continue perpetually, and we would still read those same four voltages over and again, forever. We have measured the voltage of the signal at discrete moments in time. The resulting voltage-measurement sequence is therefore called a *discrete-time signal*.

If we save each discrete-time signal voltage in a computer memory and we know the frequency at which we sampled the signal, we have a *discrete-time sampled signal*. This is what an analog-to-digital converter (ADC) does. It uses a sampling clock to measure discrete samples of an incoming analog signal at precise times, and it produces a digital representation of the input sample voltage.

In 1933, Harry Nyquist discovered that to accurately recover all the components of a periodic waveform, it is necessary to use a sampling frequency of at least twice the bandwidth of the signal being measured. That minimum sampling frequency is called the *Nyquist criterion*. This may be expressed as:

$$f_s \geq 2f_{bw} \qquad \text{(Eq 1)}$$

where f_s is the sampling rate and f_{bw} is the bandwidth. See? The math isn't so bad, is it?

Now as an example of the Nyquist criterion, let's consider human hearing, which typically ranges from 20 Hz to 20 kHz. To recreate this frequency response, a CD player must sample at a frequency of at least 40 kHz. As we will soon learn, the maximum frequency component must be limited to 20 kHz through low-pass filtering to prevent distortion caused by false images of the signal. To ease filter requirements, therefore, CD players use a standard sampling rate of 44,100 Hz. All modern PC sound cards support that sampling rate.

What happens if the sampled bandwidth is greater than half the sampling rate and is not limited by a low-pass filter? An *alias* of the signal is produced that appears in the output along with the original signal. Aliases can cause distortion, beat notes and unwanted spurious images. Fortunately, alias frequencies can be precisely predicted and prevented with proper low-pass or band-pass filters, which are often referred to as *anti-aliasing* filters, as shown in Fig 1. There are even cases where the alias frequency can be used to advantage; that will be discussed later in the article.

This is the point where most texts on DSP go into great detail about what sampled signals look like above the Nyquist frequency. Since the goal of this article is practical implementation, I refer you to Chapter 3 of the DSP Guide for a more in-depth discussion of sampling, aliases, A-to-D and

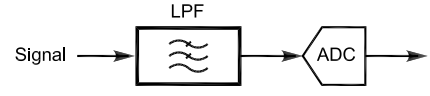


Fig 1—A/D conversion with antialiasing low-pass filter.

D-to-A conversion. Also refer to Doug Smith's article, "Signals, Samples, and Stuff: A DSP Tutorial."[1]

What you need to know for now is that if we adhere to the Nyquist criterion in Eq 1, we can accurately sample, process and recreate virtually any desired waveform. The sampled signal will consist of a series of numbers in computer memory measured at time intervals equal to the sampling rate. Since we now know the amplitude of the signal at discrete time intervals, we can process the digitized signal in software with a precision and flexibility not possible with analog circuits.

From RF to a PC's Sound Card

Our objective is to convert a modulated radio-frequency signal from the frequency domain to the time domain for software processing. In the frequency domain, we measure amplitude versus frequency (as with a spectrum analyzer); in the time domain, we measure amplitude versus time (as with an oscilloscope).

In this application, we choose to use a standard 16-bit PC sound card that has a maximum sampling rate of 44,100 Hz. According to Eq 1, this means that the maximum-bandwidth signal we can accommodate is 22,050 Hz. With quadrature sampling, discussed later, this can actually be extended to 44 kHz. Most sound cards have built-in antialiasing filters that cut off sharply at around 20 kHz. (For a couple hundred dollars more, PC sound cards are now available that support 24 bits at a 96-kHz sampling rate with up to 105 dB of dynamic range.)

Most commercial and amateur DSP designs use dedicated DSPs that sample intermediate frequencies (IFs) of 40 kHz or above. They use traditional analog superheterodyne techniques for down-conversion and filtering. With the advent of very-high-speed and wide-bandwidth ADCs, it is now possible to directly sample signals up through the entire HF range and even into the low VHF range. For example, the Analog Devices AD9430 A/D converter is specified with sample rates up to 210 Msps at 12 bits of resolution and a 700-MHz bandwidth. That 700-MHz bandwidth can be used in under-sampling applications, a topic that is beyond the scope of this article series.

The goal of my project is to build a PC-based software-defined radio that uses as little external hardware as possible while maximizing dynamic range and flexibility. To do so, we will need to convert the RF signal to audio frequencies in a way that allows removal of the unwanted mixing products or images caused by the down-conversion process. The simplest way to accomplish this while maintaining wide dynamic range is to use D-C techniques to translate the modulated RF signal directly to baseband.

[1]Notes appear on page 21.

We can mix the signal with an oscillator tuned to the RF carrier frequency to translate the bandwidth-limited signal to a 0-Hz IF as shown in Fig 2.

The example in the figure shows a 14.001-MHz carrier signal mixed with a 14.000-MHz local oscillator to translate the carrier to 1 kHz. If the low-pass filter had a cutoff of 1.5 kHz, any signal between 14.000 MHz and 14.0015 MHz would be within the passband of the direct-conversion receiver. The problem with this simple approach is that we would also simultaneously receive all signals between 13.9985 MHz and 14.000 MHz as unwanted images within the passband, as illustrated in Fig 3. Why is that?

Most amateurs are familiar with the concept of sum and difference frequencies that result from mixing two signals. When a carrier frequency, f_c, is mixed with a local oscillator, f_{lo}, they combine in the general form:

$$f_c f_{lo} = \frac{1}{2}[(f_c + f_{lo}) + (f_c - f_{lo})] \qquad \text{(Eq 2)}$$

When we use the direct-conversion mixer shown in Fig 2, we will receive these primary output signals:

$$f_c + f_{lo} = 14.001\ \text{MHz} + 14.000\ \text{MHz} = 28.001\ \text{MHz}$$
$$f_c - f_{lo} = 14.001\ \text{MHz} - 14.000\ \text{MHz} = 0.001\ \text{MHz}$$

Note that we also receive the image frequency that "folds over" the primary output signals:

$$-f_c + f_{lo} = -14.001\ \text{MHz} + 14.000\ \text{MHz} = -0.001\ \text{MHz}$$

A low-pass filter easily removes the 28.001-MHz *sum frequency*, but the –0.001-MHz *difference-frequency* image will remain in the output. This unwanted image is the lower sideband with respect to the 14.000-MHz carrier frequency. This would not be a problem if there were no signals below 14.000 MHz to interfere. As previously stated, all undesired signals between 13.9985 and 14.000 MHz will translate into the passband along with the desired signals above 14.000 MHz. The image also results in increased noise in the output.

So how can we remove the image-frequency signals? It can be accomplished through *quadrature mixing*. Phasing or quadrature transmitters and receivers—also called Weaver-method or image-rejection mixers—have existed since the early days of single sideband. In fact, my first SSB transmitter was a used Central Electronics 20A exciter that incorporated a phasing design. Phasing systems lost favor in the early 1960s with the advent of relatively inexpensive, high-performance filters.

To achieve good opposite-sideband or image suppression, phasing systems require a precise balance of amplitude and phase between two samples of the signal that are 90° out

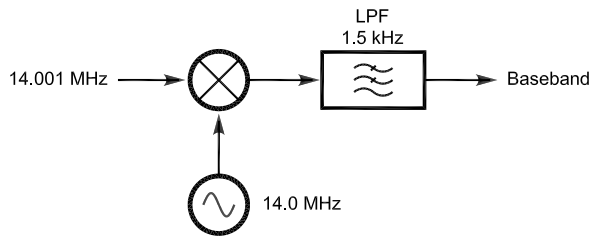


Fig 2—A direct-conversion real mixer with a 1.5-kHz low-pass filter.

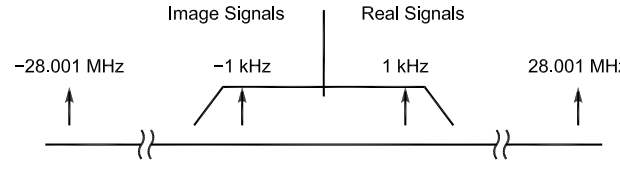


Fig 3—Output spectrum of a real mixer illustrating the sum, difference and image frequencies.

of phase or in quadrature with each other—"orthogonal" is the term used in some texts. Until the advent of digital signal processing, it was difficult to realize the level of image rejection performance required of modern radio systems in phasing designs. Since digital signal processing allows precise numerical control of phase and amplitude, quadrature modulation and demodulation are the preferred methods. Such signals in quadrature allow virtually any modulation method to be implemented in software using DSP techniques.

Give Me *I* and *Q* and I Can Demodulate Anything

First, consider the direct-conversion mixer shown in Fig 2. When the RF signal is converted to baseband audio using a single channel, we can visualize the output as varying in amplitude along a single axis as illustrated in Fig 4. We will refer to this as the *in-phase* or *I* signal. Notice that its magnitude varies from a positive value to a negative value at the frequency of the modulating signal. If we use a diode to rectify the signal, we would have created a simple envelope or AM detector.

Remember that in AM envelope detection, both modulation sidebands carry information energy and both are desired at the output. Only amplitude information is required to fully demodulate the original signal. The problem is that most other modulation techniques require that the phase of the signal be known. This is where quadrature detection comes in. If we delay a copy of the RF carrier by 90° to form a quadrature (*Q*) signal, we can then use it in conjunction with the original in-phase signal and the math we learned in middle school to determine the instantaneous phase and amplitude of the original signal.

Fig 5 illustrates an RF carrier with the level of the *I* signal plotted on the x-axis and that of the *Q* signal plotted on the y-axis of a plane. This is often referred to in the literature as a *phasor diagram* in the *complex plane*. We are now able to extrapolate the two signals to draw an arrow or phasor that represents the instantaneous magnitude and phase of the original signal.

Okay, here is where you will have to use a couple of those extra functions on the calculator. To compute the magnitude m_t or envelope of the signal, we use the geometry of right triangles. In a right triangle, the square of the hypotenuse is equal to the sum of the squares of the other two sides—according to the Pythagorean theorem. Or restating, the hypotenuse as m_t (magnitude with respect to time):

$$m_t = \sqrt{I_t^2 + Q_t^2} \qquad \text{(Eq 3)}$$

The instantaneous phase of the signal as measured counterclockwise from the positive *I* axis and may be computed by the inverse tangent (or arctangent) as follows:

$$\phi_t = \tan^{-1}\left(\frac{Q_t}{I_t}\right) \qquad \text{(Eq 4)}$$

Therefore, if we measured the instantaneous values of *I* and *Q*, we would know everything we needed to know about the signal at a given moment in time. This is true whether we are dealing with continuous analog signals or discrete sampled signals. With *I* and *Q*, we can demodulate AM signals directly using Eq 3 and FM signals using Eq 4. To demodulate SSB takes one more step. Quadrature signals can be used analytically to remove the image frequencies and leave only the desired sideband.

The mathematical equations for quadrature signals are difficult but are very understandable with a little study.[2] I highly recommend that you read the online article, "Quadrature Signals: Complex, But Not Complicated," by Richard Lyons. It can be found at **www.dspguru.com/info/tutor/quadsig.htm**. The article develops in a very logical manner how quadrature-sampling *I*/*Q* demodulation is accomplished. A basic understanding of these concepts is essential to designing software-defined radios.

We can take advantage of the analytic capabilities of quadrature signals through a quadrature mixer. To understand the basic concepts of quadrature mixing, refer to Fig 6, which illustrates a quadrature-sampling *I*/*Q* mixer.

First, the RF input signal is bandpass filtered and applied to the two parallel mixer channels. By delaying the local oscillator wave by 90°, we can generate a cosine wave that, in tandem, forms a *quadrature oscillator*. The RF carrier, $f_c(t)$, is mixed with the respective cosine and sine wave local oscillators and is subsequently low-pass filtered to create the in-phase, $I(t)$, and quadrature, $Q(t)$, signals. The $Q(t)$

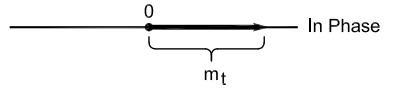


Fig 4—An in-phase signal (*I*) on the real plane. The magnitude, $m_{(t)}$, is easily measured as the instantaneous peak voltage, but no phase information is available from in-phase detection. This is the way an AM envelope detector works.

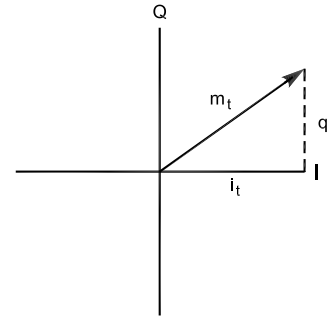


Fig 5—I + *j*Q are shown on the complex plane. The vector rotates counterclockwise at a rate of $2\pi f_c$. The magnitude and phase of the rotating vector at any instant in time may be determined through Eqs 3 and 4.

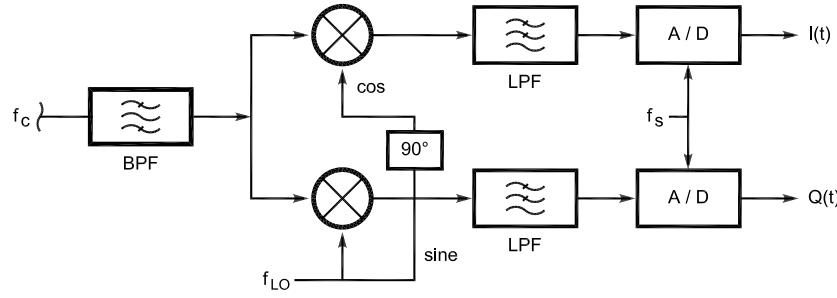


Fig 6—Quadrature sampling mixer: The RF carrier, f_c, is fed to parallel mixers. The local oscillator (Sine) is fed to the lower-channel mixer directly and is delayed by 90° (Cosine) to feed the upper-channel mixer. The low-pass filters provide antialias filtering before analog-to-digital conversion. The upper channel provides the in-phase ($I_{(t)}$) signal and the lower channel provides the quadrature ($Q_{(t)}$) signal. In the PC SDR the low-pass filters and A/D converters are integrated on the PC sound card.

channel is phase-shifted 90° relative to the $I(t)$ channel through mixing with the sine local oscillator. The low-pass filter is designed for cutoff below the Nyquist frequency to prevent aliasing in the A/D step. The A/D converts continuous-time signals to discrete-time sampled signals. Now that we have the I and Q samples in memory, we can perform the magic of digital signal processing.

Before we go further, let me reiterate that one of the problems with this method of down-conversion is that it can be costly to get good opposite-sideband suppression with analog circuits. Any variance in component values will cause phase or amplitude imbalance between two channels, resulting in a corresponding decrease in opposite-sideband suppression. With analog circuits, it is difficult to achieve better than 40 dB of suppression without much higher cost. Fortunately, it is straightforward to correct the analog imbalances in software.

Another significant drawback of direct-conversion receivers is that the noise increases as the demodulated signal approaches 0 Hz. Noise contributions come from a number of sources, such as 1/*f* noise from the semiconductor devices themselves, 60-Hz and 120-Hz line noise or hum, microphonic mechanical noise and local-oscillator phase noise near the carrier frequency. This can limit sensitivity since most people prefer their CW tones to be below 1 kHz. It turns out that most of the low-frequency noise rolls off above 1 kHz. Since a sound card can process signals all the way up to 20 kHz, why not use some of that bandwidth to move away from the low frequency noise? The PC SDR uses an 11.025-kHz, *offset-baseband IF* to reduce the noise to a manageable level. By offsetting the local oscillator by 11.025 kHz, we can now receive signals near the carrier frequency without any of the low-frequency noise issues. This also significantly reduces the effects of local-oscillator phase noise. Once we have digitally captured the signal, it is a trivial software task to shift the demodulated signal down to a 0-Hz offset.

DSP in the Frequency Domain

Every DSP text I have read thus far concentrates on time-domain filtering and demodulation of SSB signals using *finite-impulse-response (FIR)* filters. Since these techniques have been thoroughly discussed in the literature[1, 3, 4] and are not currently used in my PC SDR, they will not be covered in this article series.

My PC SDR uses the power of the *fast Fourier transform (FFT)* to do almost all of the heavy lifting in the frequency domain. Most DSP texts use a lot of ink to derive the math so that one can write the FFT code. Since Intel has so helpfully provided the code in executable form in their signal-processing library,[5] we don't care how to write an FFT: We just need to know how to use it. Simply put, the FFT converts the complex I and Q discrete-time signals into the frequency domain. The FFT output can be thought of as a large bank of very narrow band-pass filters, called *bins*, each one measuring the spectral energy within its respective bandwidth. The output resembles a *comb filter* wherein each bin slightly overlaps its adjacent bins forming a scalloped curve, as shown in Fig 7. When a signal is precisely at the center frequency of a bin, there will be a corresponding value only in that bin. As the frequency is offset from the bin's center, there will be a corresponding increase in the value of the adjacent bin and a decrease in the value of the current bin. Mathematical analysis fully describes the relationship between FFT bins,[6] but such is beyond the scope of this article.

Further, the FFT allows us to measure both phase and amplitude of the signal within each bin using Eqs 3 and 4 above. The complex version allows us to measure positive and negative frequencies separately. Fig 8 illustrates the output of a complex, or quadrature, FFT.

The bandwidth of each FFT bin may be computed as shown in Eq 5, where BW_{bin} is the bandwidth of a single bin, f_s is the sampling rate and N is the size of the FFT. The center frequency of each FFT bin may be determined by Eq 6 where f_{center} is the bin's center frequency, n is the bin number, f_s is the sampling rate and N is the size of the FFT. Bins zero through $(N/2)-1$ represent upper-sideband frequencies and bins $N/2$ to $N-1$ represent lower-sideband frequencies around the carrier frequency.

$$BW_{bin} = \frac{f_s}{N} \quad \text{(Eq 5)}$$

$$f_{center} = \frac{nf_s}{N} \quad \text{(Eq 6)}$$

If we assume the sampling rate of the sound card is 44.1 kHz and the number of FFT bins is 4096, then the bandwidth and center frequency of each bin would be:

$$BW_{bin} = \frac{44100}{4096} = 10.7666 \text{ Hz and}$$

$$f_{center} = n10.7666 \text{ Hz}$$

What this all means is that the receiver will have 4096, ~11-Hz-wide

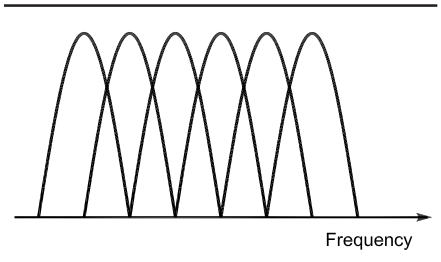


Fig 7—FFT output resembles a comb filter: Each bin of the FFT overlaps its adjacent bins just as in a comb filter. The 3-dB points overlap to provide linear output. The phase and magnitude of the signal in each bin is easily determined mathematically with Eqs 3 and 4.

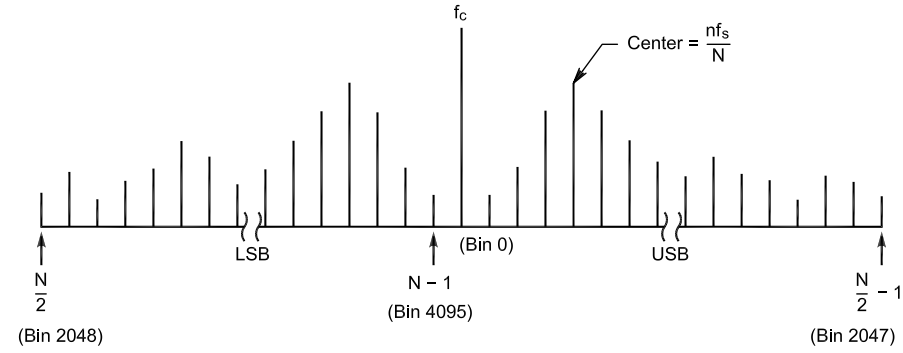


Fig 8—Complex FFT output: The output of a complex FFT may be thought of as a series of band-pass filters aligned around the carrier frequency, f_c, at bin 0. *N* represents the number of FFT bins. The upper sideband is located in bins 1 through *(N*/2)–1 and the lower sideband is located in bins *N*/2 to *N*–1. The center frequency and bandwidth of each bin may be calculated using Eqs 5 and 6.

band-pass filters. We can therefore create band-pass filters from 11 Hz to approximately 40 kHz in 11-Hz steps.

The PC SDR performs the following functions in the frequency domain after FFT conversion:

- Brick-wall fixed and variable band-pass filters
- Frequency conversion
- SSB/CW demodulation
- Sideband selection
- Frequency-domain noise subtraction
- Frequency-selective squelch
- Noise blanking
- Graphic equalization ("tone control")
- Phase and amplitude balancing to remove images
- SSB generation
- Future digital modes such as PSK31 and RTTY

Once the desired frequency-domain processing is completed, it is simple to convert the signal back to the time domain by using an *inverse FFT*. In the PC SDR, only AGC and adaptive noise filtering are currently performed in the time domain. A simplified diagram of the PC SDR software architecture is provided in Fig 9. These concepts will be discussed in detail in a future article.

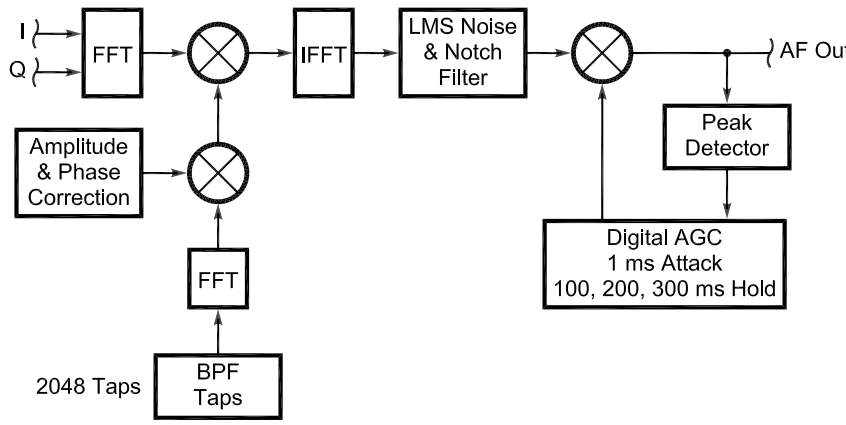


Fig 9—SDR receiver software architecture: The *I* and *Q* signals are fed from the sound-card input directly to a 4096-bin complex FFT. Band-pass filter coefficients are precomputed and converted to the frequency domain using another FFT. The frequency-domain filter is then multiplied by the frequency-domain signal to provide brick-wall filtering. The filtered signal is then converted to the time domain using the inverse FFT. Adaptive noise and notch filtering and digital AGC follow in the time domain.

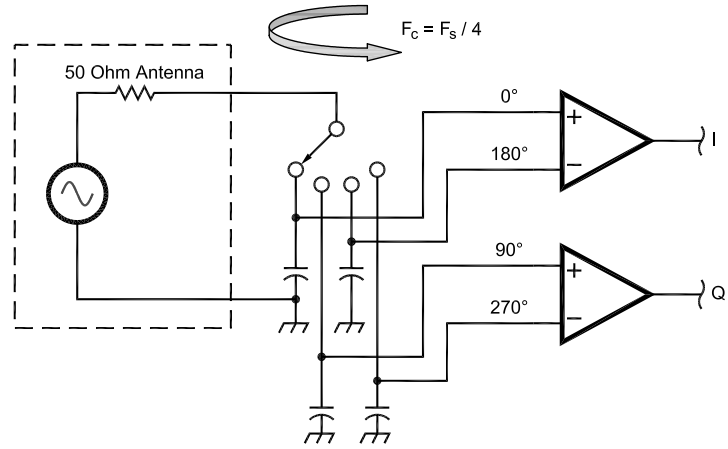


Fig 10—Tayloe detector: The switch rotates at the carrier frequency so that each capacitor samples the signal once each revolution. The 0° and 180° capacitors differentially sum to provide the in-phase (*I*) signal and the 90° and 270° capacitors sum to provide the quadrature (*Q*) signal.

Sampling RF Signals with the Tayloe Detector: A New Twist on an Old Problem

While searching the Internet for information on quadrature mixing, I ran across a most innovative and elegant design by Dan Tayloe, N7VE. Dan, who works for Motorola, has developed and patented (US Patent #6,230,000) what has been called the *Tayloe detector*.[7] The beauty of the Tayloe detector is found in both its design elegance and its exceptional performance. It resembles other concepts in design, but appears unique in its high performance with minimal components.[8, 9, 10, 11] In its simplest form, you can build a complete quadrature down converter with only three or four ICs (less the local oscillator) at a cost of less than $10.

Fig 10 illustrates a single-balanced version of the Tayloe detector. It can be visualized as a four-position rotary switch revolving at a rate equal to the carrier frequency. The 50-Ω antenna impedance is connected to the rotor and each of the four switch positions is connected to a *sampling capacitor*. Since the switch rotor is turning at exactly the RF carrier frequency, each capacitor will track the carrier's amplitude for exactly one-quarter of the cycle and will hold its value for the remainder of the cycle. The rotating switch will therefore sample the signal at 0°, 90°, 180° and 270°, respectively.

As shown in Fig 11, the 50-Ω impedance of the antenna and the sampling capacitors form an R-C low-pass filter during the period when each respective switch is turned on. Therefore, each sample represents the integral or average voltage of the signal during its respective one-quarter cycle. When the switch is off, each sampling capacitor will hold its value until the next revolution. If the RF carrier and the rotating frequency were exactly in phase, the output of each capacitor will be a dc level equal to the average

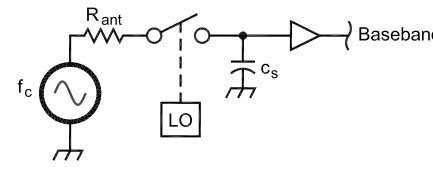


Fig 11—Track and hold sampling circuit: Each of the four sampling capacitors in the Tayloe detector form an RC track-and-hold circuit. When the switch is on, the capacitor will charge to the average value of the carrier during its respective one-quarter cycle. During the remaining three-quarters cycle, it will hold its charge. The local-oscillator frequency is equal to the carrier frequency so that the output will be at baseband.

value of the sample.

If we differentially sum outputs of the 0° and 180° sampling capacitors with an op amp (see Fig 10), the output would be a dc voltage equal to two times the value of the individually sampled values when the switch rotation frequency equals the carrier frequency. Imagine, 6 dB of noise-free gain! The same would be true for the 90° and 270° capacitors as well. The 0°/180° summation forms the *I* channel and the 90°/270° summation forms the *Q* channel of the quadrature down-conversion.

As we shift the frequency of the carrier away from the sampling frequency, the values of the inverting phases will no longer be dc levels. The output frequency will vary according to the "beat" or difference frequency between the carrier and the switch-rotation frequency to provide an accurate representation of all the signal components converted to baseband.

Fig 12 provides the schematic for a simple, single-balanced Tayloe detector. It consists of a PI5V331, 1:4 FET demultiplexer that switches the signal to each of the four sampling capacitors. The 74AC74 dual flip-flop is connected as a divide-by-four Johnson counter to provide the two-phase clock to the demultiplexer chip. The outputs of the sampling capacitors are differentially summed through the two LT1115 ultra-low-noise op amps to form the *I* and *Q* outputs, respectively. Note that the impedance of the antenna forms the input resistance for the op-amp gain as shown in Eq 7. This impedance may vary significantly with the actual antenna. I use instrumentation amplifiers in my final design to eliminate gain variance with antenna impedance. More information on the hardware design will be provided in a future article.

Since the duty cycle of each switch is 25%, the effective resistance in the RC network is the antenna impedance multiplied by four in the op-amp gain formula, as shown in Eq 7:

$$G = \frac{R_f}{4R_{ant}} \qquad \text{(Eq 7)}$$

For example, with a feedback resistance, R_f, of 3.3 kΩ and antenna impedance, R_{ant}, of 50 Ω, the resulting gain of the input stage is:

$$G = \frac{3300}{4 \times 50} = 16.5$$

The Tayloe detector may also be analyzed as a *digital commutating filter*.[12, 13, 14] This means that it operates as a very-high-*Q* tracking filter, where Eq 8 determines the bandwidth and *n* is the number of sampling capacitors,

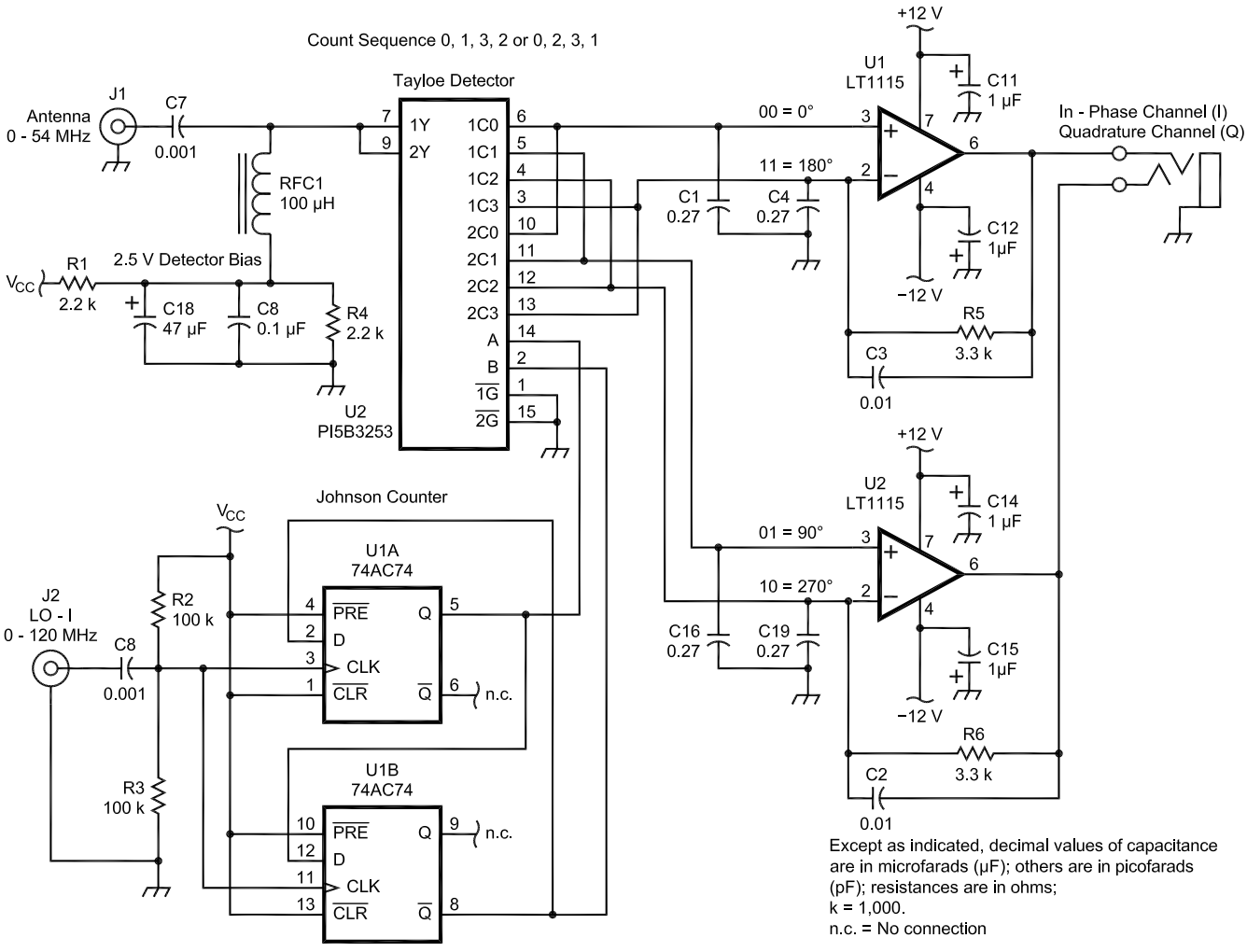


Fig 12—Singly balanced Tayloe detector.

R_{ant} is the antenna impedance and C_s is the value of the individual sampling capacitors. Eq 9 determines the Q_{det} of the filter, where f_c is the center frequency and BW_{det} is the bandwidth of the filter.

$$BW_{det} = \frac{1}{\pi n R_{ant} C_s} \quad \text{(Eq 8)}$$

$$Q_{det} = \frac{f_c}{BW_{det}} \quad \text{(Eq 9)}$$

By example, if we assume the sampling capacitor to be 0.27 µF and the antenna impedance to be 50 Ω, then BW and Q are computed as follows:

$$BW_{det} = \frac{1}{(\pi)(4)(50)(2.7\times10^{-7})} = 5895 \text{ Hz}$$

$$Q_{det} = \frac{14.001\times10^{6}}{5895} = 2375$$

Since the PC SDR uses an offset baseband IF, I have chosen to design the detector's bandwidth to be 40 kHz to allow low-frequency noise elimination as discussed above.

The real payoff in the Tayloe detector is its performance. It has been stated that the *ideal* commutating mixer has a minimum conversion loss (which equates to noise figure) of 3.9 dB.[15, 16] Typical high-level diode mixers have a conversion loss of 6-7 dB and noise figures 1 dB higher than the loss. The Tayloe detector has less than 1 dB of conversion loss, remarkably. How can this be? The reason is that it is not really a mixer but a sampling detector in the form of a quadrature track and hold. This means that the design adheres to discrete-time sampling theory, which, while similar to mixing, has its own unique characteristics. Because a track and hold actually holds the signal value between samples, the signal output never goes to zero.

This is where aliasing can actually be used to our benefit. Since each switch and capacitor in the Tayloe detector actually samples the RF signal once each cycle, it will respond to alias frequencies as well as those within the Nyquist frequency range. In a traditional direct-conversion receiver, the local-oscillator frequency is set to the carrier frequency so that the difference frequency, or IF, is at 0 Hz and the sum frequency is at two times the carrier frequency per Eq 2. We normally remove the sum frequency through low-pass filtering, resulting in conversion loss and a corresponding increase in noise figure. In the Tayloe detector, the sum frequency resides at the first alias frequency as shown in Fig 13. Remember that an alias is a real signal and will appear in the output as if it were a baseband signal. Therefore, the alias adds to the baseband signal for a theoretically lossless detector. In real life, there is a slight loss due to the resistance of the switch and aperture loss due to imperfect switching times.

PC SDR Transceiver Hardware

The Tayloe detector therefore provides a low-cost, high-performance method for both quadrature down-conversion as well as up-conversion for transmitting. For a complete system, we would need to provide analog AGC to prevent overload of the ADC inputs and a means of digital frequency control. Fig 14 illustrates the hardware architecture of the PC SDR receiver as it currently exists. The challenge has been to build a low-noise analog chain that matches the dynamic range of the Tayloe detector to the dynamic range of the PC sound card. This will be covered in a future article.

I am currently prototyping a complete PC SDR transceiver, the SDR-1000, that will provide general-coverage receive from 100 kHz to 54 MHz and will transmit on all ham bands from 160 through 6 meters.

SDR Applications

At the time of this writing, the typical entry-level PC now runs at a clock frequency greater than 1 GHz and costs only a few hundred dollars. We now have exceptional processing power at our disposal to perform DSP tasks that were once only dreams. The transfer of knowledge from the aca-

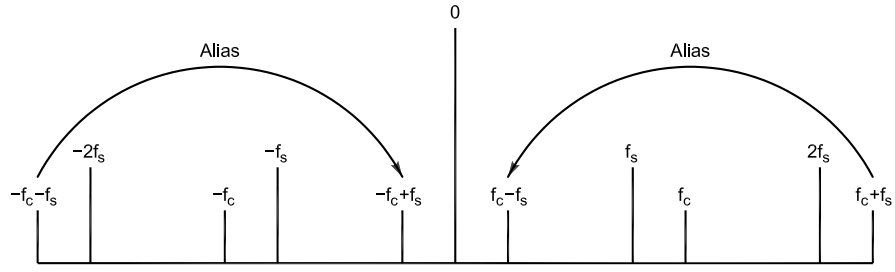


Fig 13—Alias summing on Tayloe detector output: Since the Tayloe detector samples the signal the sum frequency (*f* c + *f* s) and its image (–*f* c – *f* s) are located at the first alias frequency. The alias signals sum with the baseband signals to eliminate the mixing product loss associated with traditional mixers. In a typical mixer, the sum frequency energy is lost through filtering thereby increasing the noise figure of the device.

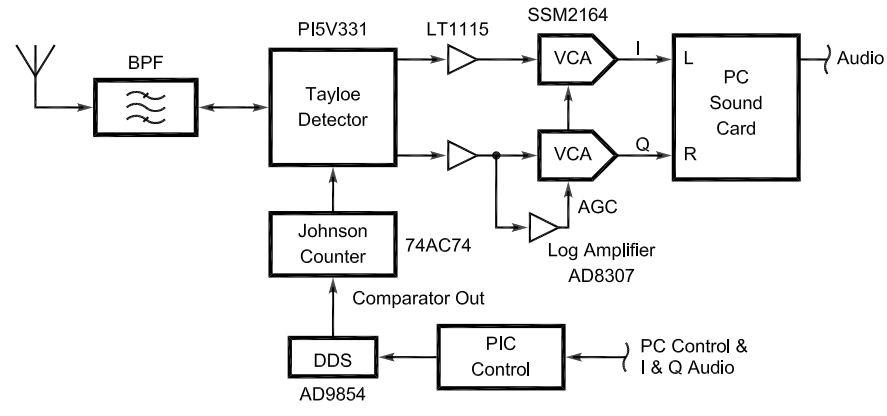


Fig 14—PC SDR receiver hardware architecture: After band-pass filtering the antenna is fed directly to the Tayloe detector, which in turn provides *I* and *Q* outputs at baseband. A DDS and a divide-by-four Johnson counter drive the Tayloe detector demultiplexer. The LT1115s offer ultra-low noise-differential summing and amplification prior to the wide-dynamic-range analog AGC circuit formed by the SSM2164 and AD8307 log amplifier.

demic to the practical is the primary limit of the availability of this technology to the Amateur Radio experimenter. This article series attempts to demystify some of the fundamental concepts to encourage experimentation within our community. The ARRL recently formed a SDR Working Group for supporting this effort, as well.

The SDR mimics the analog world in digital data, which can be manipulated much more precisely. Analog radio has always been modeled mathematically and can therefore be processed in a computer. This means that virtually any modulation scheme may be handled digitally with performance levels difficult, or impossible, to attain with analog circuits. Let's consider some of the amateur applications for the SDR:

- Competition-grade HF transceivers
- High-performance IF for microwave bands
- Multimode digital transceiver
- EME and weak-signal work
- Digital-voice modes
- Dream it and code it

For Further Reading

For more in-depth study of DSP techniques, I highly recommend that you purchase the following texts in order of their listing:

Understanding Digital Signal Processing by Richard G. Lyons (see Note 6). This is one of the best-written textbooks about DSP.

Digital Signal Processing Technology by Doug Smith (see Note 4). This new book explains DSP theory and application from an Amateur Radio perspective.

Digital Signal Processing in Communications Systems by Marvin E. Frerking (see Note 3). This book relates DSP theory specifically to modulation and demodulation techniques for radio applications.

Acknowledgements

I would like to thank those who have assisted me in my journey to understanding software radios. Dan Tayloe, N7VE, has always been helpful and responsive in answering questions about the Tayloe detector. Doug Smith, KF6DX, and Leif Åsbrink, SM5BSZ, have been gracious to answer my questions about DSP and receiver design on numerous occasions. Most of all, I want to thank my Saturday-morning breakfast review team: Mike Pendley, WA5VTV; Ken Simmons, K5UHF; Rick Kirchhof, KD5ABM; and Chuck McLeavy, WB5BMH. These guys put up with my questions every week and have given me tremendous advice and feedback all throughout the project. I also want to thank my wonderful wife, Virginia, who has been incredibly patient with all the hours I have put in on this project.

Where Do We Go From Here?

Three future articles will describe the construction and programming of the PC SDR. The next article in the series will detail the software interface to the PC sound card. Integrating full-duplex sound with *DirectX* was one of the more challenging parts of the project. The third article will describe the *Visual Basic* code and the use of the Intel Signal Processing Library for implementing the key DSP algorithms in radio communications. The final article will describe the completed transceiver hardware for the SDR-1000.

Notes

[1]D. Smith, KF6DX, "Signals, Samples and Stuff: A DSP Tutorial (Part 1)," *QEX,* Mar/Apr 1998, pp 3-11.

[2]J. Bloom, KE3Z, "Negative Frequencies and Complex Signals," *QEX*, Sep 1994, pp 22-27.

[3]M. E. Frerking, *Digital Signal Processing in Communication Systems* (New York: Van Nostrand Reinhold, 1994, ISBN: 0442016166), pp 272-286.

[4]D. Smith, KF6DX, *Digital Signal Processing Technology* (Newington, Connecticut: ARRL, 2001), pp 5-1 through 5-38.

[5]The Intel Signal Processing Library is available for download at **developer.intel.com/software/products/perflib/spl/**.

[6]R. G. Lyons, *Understanding Digital Signal Processing*, (Reading, Massachusetts: Addison-Wesley, 1997), pp 49-146.

[7]D. Tayloe, N7VE, "Letters to the Editor, Notes on'Ideal' Commutating Mixers (Nov/Dec 1999)," *QEX,* March/April 2001, p 61.

[8]P. Rice, VK3BHR, "SSB by the Fourth Method?" available at **ironbark.bendigo.latrobe.edu.au/~rice/ssb/ssb.html**.

[9]A. A. Abidi, "Direct-Conversion Radio Transceivers for Digital Communications," *IEEE Journal of Solid-State Circuits*, Vol 30, No 12, December 1995, pp 1399-1410, Also on the Web at **www.icsl.ucla.edu/aagroup/PDF_files/dir-con.pdf**

[10]P. Y. Chan, A. Rofougaran, K.A. Ahmed, and A. A. Abidi, "A Highly Linear 1-GHz CMOS Downconversion Mixer." Presented at the European Solid State Circuits Conference, Seville, Spain, Sep 22-24, 1993, pp 210-213 of the conference proceedings. Also on the Web at **www.icsl.ucla.edu/aagroup/PDF_files/mxr-93.pdf**

[11]D. H. van Graas, PA0DEN, "The Fourth Method: Generating and Detecting SSB Signals," *QEX*, Sep 1990, pp 7-11. This circuit is very similar to a Tayloe detector, but it has a lot of unnecessary components.

[12]M. Kossor, WA2EBY, "A Digital Commutating Filter," *QEX,* May/Jun 1999, pp 3-8.

[13]C. Ping, BA1HAM, "An Improved Switched Capacitor Filter," *QEX,* Sep/Oct 2000, pp 41-45.

[14]P. Anderson, KC1HR, "Letters to the Editor, A Digital Commutating Filter," *QEX,* Jul/Aug 1999, pp 62.

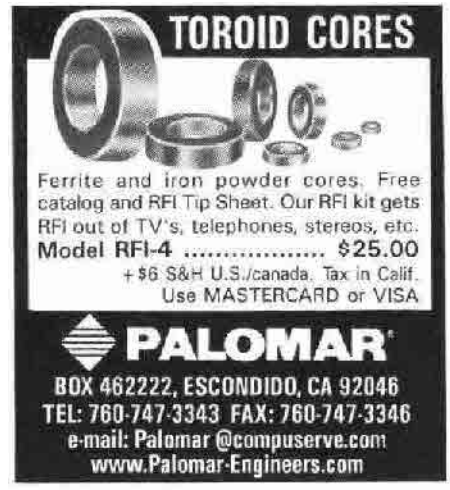
[15]D. Smith, KF6DX, "Notes on 'Ideal' Commutating Mixers," *QEX,* Nov/Dec 1999, pp 52-54.

[16]P. Chadwick, G3RZP, "Letters to the Editor, Notes on 'Ideal' Commutating Mixers" (Nov/Dec 1999), *QEX,* Mar/Apr 2000, pp 61-62.

Gerald became a ham in 1967 during high school, first as a Novice and then a General class as WA5RXV. He completed his Advanced class license and became KE5OH before finishing high school and received his First Class Radiotelephone license while working in the television broadcast industry during college. After 25 years of inactivity, Gerald returned to the active amateur ranks in 1997 when he completed the requirements for Extra class license and became AC5OG.

Gerald lives in Austin, Texas, and is currently CEO of Sixth Market Inc, a hedge fund that trades equities using artificial-intelligence software. Gerald previously founded and ran five technology companies spanning hardware, software and electronic manufacturing. Gerald holds a Bachelor of Science Degree in Electrical Engineering from Mississippi Stage University.

Gerald is a member of the ARRL SDR working Group and currently enjoys homebrew software-radio development, 6-meter DX and satellite operations. □□


TOROID CORES
Ferrite and iron powder cores. Free catalog and RFI Tip Sheet. Our RFI kit gets RFI out of TV's, telephones, stereos, etc.
Model RFI-4 ..... $25.00
+$6 S&H U.S./canada. Tax in Calif.
Use MASTERCARD or VISA
PALOMAR
BOX 462222, ESCONDIDO, CA 92046
TEL: 760-747-3343 FAX: 760-747-3346
e-mail: Palomar @compuserve.com
www.Palomar-Engineers.com

A Software-Defined Radio for the Masses, Part 2

Come learn how to use a PC sound card to enter the wonderful world of digital signal processing.

By Gerald Youngblood, AC5OG

Part 1 gave a general description of digital signal processing (DSP) in software-defined radios (SDRs).[1] It also provided an overview of a full-featured radio that uses a personal computer to perform all DSP functions. This article begins design implementation with a complete description of software that provides a full-duplex interface to a standard PC sound card.

To perform the magic of digital signal processing, we must be able to convert a signal from analog to digital and back to analog again. Most amateur experimenters already have this capability in their shacks and many have used it for slow-scan television or the new digital modes like PSK31.

Part 1 discussed the power of quadrature signal processing using *in-phase* (*I*) and *quadrature* (*Q*) signals to receive or transmit using virtually any modulation method. Fortunately, all modern PC sound cards offer the perfect method for digitizing the *I* and *Q* signals. Since virtually all cards today provide 16-bit stereo at 44-kHz sampling rates, we have exactly what we need capture and process the signals in software. Fig 1 illustrates a direct quadrature-conversion mixer connection to a PC sound card.

This article discusses complete source code for a DirectX sound-card interface in Microsoft *Visual Basic*. Consequently, the discussion assumes that the reader has some fundamental knowledge of high-level language programming.

Sound Card and PC Capabilities

Very early PC sound cards were low-performance, 8-bit mono versions. Today, virtually all PCs come with 16-bit stereo cards of sufficient quality to be used in a software-defined radio. Such a card will allow us to demodulate, filter and display up to approximately a 44-kHz bandwidth, assuming a 44-kHz sampling rate. (The bandwidth is 44 kHz, rather than 22 kHz, because the use of two channels effectively doubles the sampling rate—*Ed.*) For high-performance applications, it is important to select a card that offers a high dynamic range—on the order of 90 dB. If you are just getting started, most PC sound cards will allow you to begin experimentation, although they

[1]Notes appear on page 18.

8900 Marybank Dr
Austin, TX 78750
gerald@sixthmarket.com

may offer lower performance.

The best 16-bit price-to-performance ratio I have found at the time of this article is the Santa Cruz 6-channel DSP Audio Accelerator from Turtle Beach Inc (**www.tbeach.com**). It offers four 18-bit internal analog-to-digital (A/D) input channels and six 20-bit digital-to-analog (D/A) output channels with sampling rates up to 48 kHz. The manufacturer specifies a 96-dB signal-to-noise ratio (SNR) and better than –91 dB total harmonic distortion plus noise (THD+N). Crosstalk is stated to be –105 dB at 100 Hz. The Santa Cruz card can be purchased from online retailers for under $70.

Each bit on an A/D or D/A converter represents 6 dB of dynamic range, so a 16-bit converter has a theoretical limit of 96 dB. A very good converter with low-noise design is required to achieve this level of performance. Many 16-bit sound cards provide no more than 12-14 effective bits of dynamic range. To help achieve higher performance, the Santa Cruz card uses an 18-bit A/D converter to deliver the 96 dB dynamic range (16-bit) specification.

A SoundBlaster 64 also provides reasonable performance on the order of 76 dB SNR according to PC AV Tech at **www.pcavtech.com**. I have used this card with good results, but I much prefer the Santa Cruz card.

The processing power needed from the PC depends greatly on the signal processing required by the application. Since I am using very-high-performance filters and large fast-Fourier transforms (FFTs), my applications require at least a 400-MHz Pentium II processor with a minimum of 128 MB of RAM. If you require less performance from the software, you can get by with a much slower machine. Since the entry level for new PCs is now 1 GHz, many amateurs have ample processing power available.

Microsoft DirectX versus Windows Multimedia

Digital signal processing using a PC sound card requires that we be able to capture blocks of digitized *I* and *Q* data through the stereo inputs, process those signals and return them to the sound-card outputs in pseudo real time. This is called *full duplex*. Unfortunately, there is no high-level software interface that offers the capabilities we need for the SDR application.

Microsoft now provides two application programming interfaces[2] (APIs) that allow direct access to the sound card under *C++* and *Visual Basic*. The original interface is the Windows Multimedia system using the Waveform Audio API. While my early work was done with the Waveform Audio API, I later abandoned it for the higher performance and simpler interface DirectX offers. The only limitation I have found with DirectX is that it does not currently support sound cards with more than 16-bits of resolution. For 24-bit cards, Windows Multimedia is required. While the Santa Cruz card supports 18-bits internally, it presents only 16-bits to the interface. For information on where to download the DirectX software development kit (SDK) see Note 2.

Circular Buffer Concepts

A typical full-duplex PC sound card allows the simultaneous capture and playback of two or more audio channels (stereo). Unfortunately, there is no high-level code in *Visual Basic* or C++ to directly support full duplex as required in an SDR. We will therefore have to write code to directly control the card through the DirectX API.

DirectX internally manages all low-level buffers and their respective interfaces to the sound-card hardware. Our code will have to manage the high-level DirectX buffers (called DirectSoundBuffer and DirectSoundCaptureBuffer) to provide uninterrupted operation in a multitasking system. The DirectSoundCaptureBuffer stores the digitized signals from the stereo

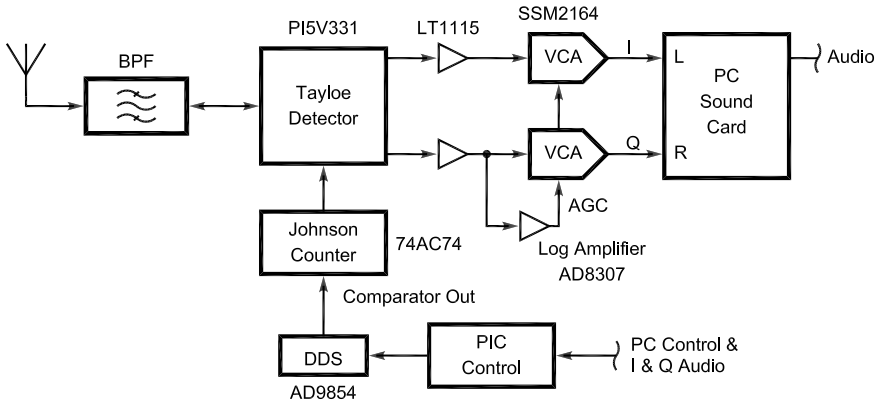


Fig 1—Direct quadrature conversion mixer to sound-card interface used in the author's prototype.

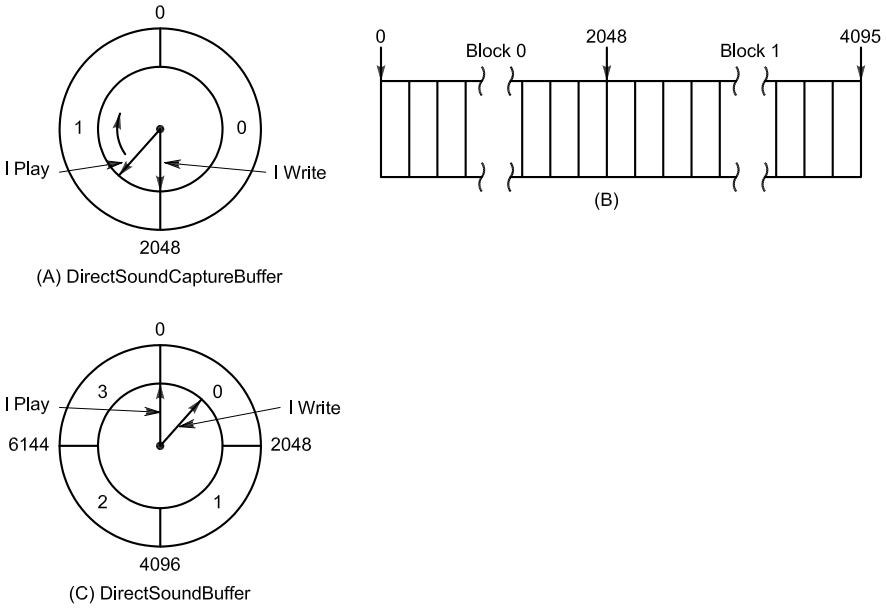


Fig 2—DirectSoundCaptureBuffer and DirectSoundBuffer circular buffer layout.

A/D converter in a circular buffer and notifies the application upon the occurrence of predefined events. Once captured in the buffer, we can read the data, perform the necessary modulation or demodulation functions using DSP and send the data to the DirectSoundBuffer for D/A conversion and output to the speakers or transmitter.

To provide smooth operation in a multitracking system without audio popping or interruption, it will be necessary to provide a multilevel buffer for both capture and playback. You may have heard the term double buffering. We will use double buffering in the DirectSoundCaptureBuffer and quadruple buffering in the DirectSoundBuffer. I found that the quad buffer with overwrite detection was required on the output to prevent overwriting problems when the system is heavily loaded with other applications. Figs 2A and 2B illustrate the concept of a circular double buffer, which is used for the DirectSoundCaptureBuffer. Although the buffer is really a linear array in memory, as shown in Fig 2B, we can visualize it as circular, as illustrated in Fig 2A. This is so because DirectX manages the buffer so that as soon as each cursor reaches the end of the array, the driver resets the cursor to the beginning of the buffer.

The DirectSoundCaptureBuffer is broken into two blocks, each equal in size to the amount of data to be captured and processed between each event. Note that an event is much like an interrupt. In our case, we will use a block size of 2048 samples. Since we are using a stereo (two-channel) board with 16 bits per channel, we will be capturing 8192 bytes per block (2048 samples × 2 channels × 2 bytes). Therefore, the DirectSoundCaptureBuffer will be twice as large (16,384 bytes).

Since the DirectSoundCapture Buffer is divided into two data blocks, we will need to send an event notification to the application after each block has been captured. The DirectX driver maintains cursors that track the position of the capture operation at all times. The driver provides the means of setting specific locations within the buffer that cause an event to trigger, thereby telling the application to retrieve the data. We may then read the correct block directly from the DirectSoundCaptureBuffer segment that has been completed.

Referring again to Fig 2A, the two cursors resemble the hands on a clock face rotating in a clockwise direction. The capture cursor, lPlay, represents the point at which data are currently being captured. (I know that sounds backward, but that is how Microsoft defined it.) The read cursor, lWrite, trails the capture cursor and indicates the point up to which data can safely be read. The data after lWrite and up to and including lPlay are not necessarily good data because of hardware buffering. We can use the lWrite cursor to trigger an event that tells the software to read each respective block of data, as will be discussed later in the article. We will therefore receive two events per revolution of the circular buffer. Data can be captured into one half of the buffer while data are being read from the other half.

Fig 2C illustrates the DirectSoundBuffer, which is used to output data to the D/A converters. In this case, we will use a quadruple buffer to allow plenty of room between the currently playing segment and the segment being written. The play cursor, lPlay, always points to the next byte of data to be played. The write cursor, lWrite, is the point after which it is safe to write data into the buffer. The cursors may be thought of as rotating in a clockwise motion just as the capture cursors do. We must monitor the location of the cursors before writing to buffer locations between the cursors to prevent overwriting data that have already been committed to the hardware for playback.

Now let's consider how the data maps from the DirectSoundCaptureBuffer to the DirectSoundBuffer. To prevent gaps or pops in the sound due to processor loading, we will want to fill the entire quadruple buffer before starting the playback looping. DirectX allows the application to set the starting point for the lPlay cursor and to start the playback at any time. Fig 3 shows how the data blocks map sequentially from the DirectSoundCaptureBuffer to the DirectSoundBuffer. Block 0 from the DirectSoundCaptureBuffer is transferred to Block 0 of the DirectSoundBuffer. Block 1 of the DirectSoundCaptureBuffer is next transferred to Block 1 of the DirectSoundBuffer and so forth. The subsequent source-code examples show how control of the buffers is accomplished.

Full Duplex, Step-by-Step

The following sections provide a detailed discussion of full-duplex DirectX implementation. The example code captures and plays back a stereo audio signal that is delayed by four

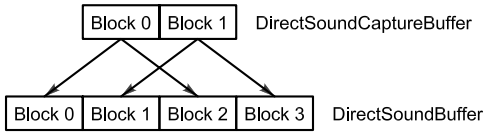


Fig 3—Method for mapping the DirectSoundCaptureBuffer to the DirectSoundBuffer.

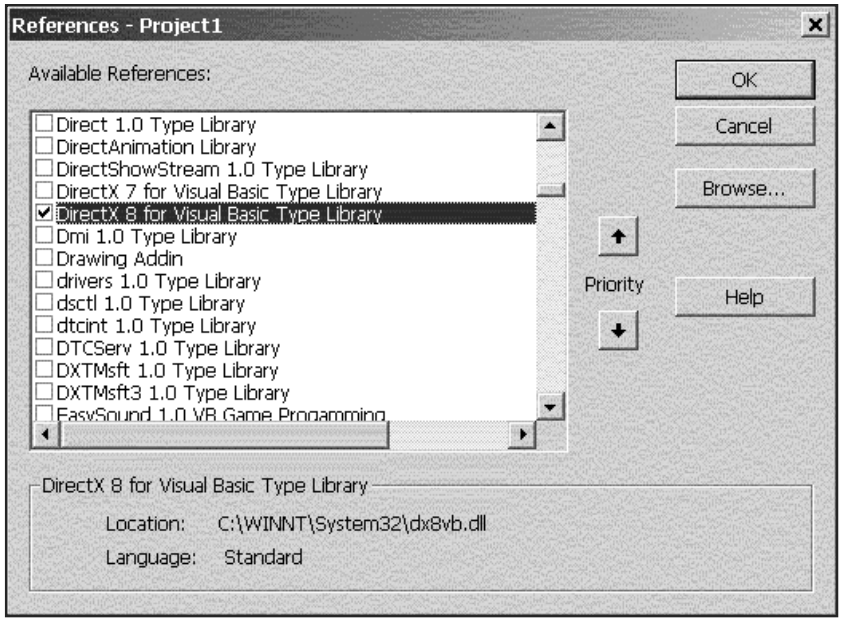


Fig 4—Registration of the DirectX8 for *Visual Basic* Type Library in the *Visual Basic* IDE.

capture periods through buffering. You should refer to the "DirectX Audio" section of the *DirectX 8.0 Programmers Reference* that is installed with the DirectX software developer's kit (SDK) throughout this discussion. The DSP code will be discussed in the next article of this series, which will discuss the modulation and demodulation of quadrature signals in the SDR. Here are the steps involved in creating the DirectX interface:

- Install DirectX runtime and SDK.
- Add a reference to DirectX8 for *Visual Basic* Type Library.
- Define Variables, I/O buffers and DirectX objects.
- Implement DirectX8 events and event handles.
- Create the audio devices.
- Create the DirectX events.
- Start and stop capture and play buffers.
- Process the DirectXEvent8.
- Fill the play buffer before starting playback.
- Detect and correct overwrite errors.
- Parse the stereo buffer into *I* and *Q* signals.
- Destroy objects and events on exit.

Complete functional source code for the DirectX driver written in Microsoft *Visual Basic* is provided for download from the *QEX* Web site.[3]

Install DirectX and Register it within Visual Basic

The first step is to download the DirectX driver and the DirectX SDK

```
Option Explicit

'Define Constants
Const Fs As Long = 44100          'Sampling frequency Hz
Const NFFT As Long = 4096         'Number of FFT bins
Const BLKSIZE As Long = 2048      'Capture/play block size
Const CAPTURESIZE As Long = 4096        'Capture Buffer size

'Define DirectX Objects
Dim dx As New DirectX8            'DirectX object
Dim ds As DirectSound8            'DirectSound object
Dim dspb As DirectSoundPrimaryBuffer8   'Primary buffer object
Dim dsc As DirectSoundCapture8    'Capture object
Dim dsb As DirectSoundSecondaryBuffer8  'Output Buffer object
Dim dscb As DirectSoundCaptureBuffer8   'Capture Buffer object

'Define Type Definitions
Dim dscbd As DSCBUFFERDESC        'Capture buffer description
Dim dsbd As DSBUFFERDESC          'DirectSound buffer description
Dim dspbd As WAVEFORMATEX         'Primary buffer description
Dim CapCurs As DSCURSORS          'DirectSound Capture Cursor
Dim PlyCurs As DSCURSORS          'DirectSound Play Cursor

'Create I/O Sound Buffers
Dim inBuffer(CAPTURESIZE) As Integer    'Demodulator Input Buffer
Dim outBuffer(CAPTURESIZE) As Integer   'Demodulator Output Buffer

'Define pointers and counters
Dim Pass As Long                  'Number of capture passes
Dim InPtr As Long                 'Capture Buffer block pointer
Dim OutPtr As Long                'Output Buffer block pointer
Dim StartAddr As Long             'Buffer block starting address
Dim EndAddr As Long               'Ending buffer block address
Dim CaptureBytes As Long          'Capture bytes to read

'Define loop counter variables for timing the capture event cycle
Dim TimeStart As Double           'Start time for DirectX8Event loop
Dim TimeEnd As Double             'Ending time for DirectX8Event loop
Dim AvgCtr As Long                'Counts number of events to average
Dim AvgTime As Double             'Stores the average event cycle time

'Set up Event variables for the Capture Buffer
Implements DirectXEvent8          'Allows DirectX Events
Dim hEvent(1) As Long             'Handle for DirectX Event
Dim EVNT(1) As DSBPOSITIONNOTIFY        'Notify position array
Dim Receiving As Boolean          'In Receive mode if true
Dim FirstPass As Boolean          'Denotes first pass from Start
```

Fig 5—Declaration of variables, buffers, events and objects. This code is located in the General section of the module or form.

from the Microsoft Web site (see Note 3). Once the driver and SDK are installed, you will need to register the DirectX8 for *Visual Basic* Type Library within the *Visual Basic* development environment.

If you are building the project from scratch, first create a *Visual Basic* project and name it "Sound." When the project loads, go to the Project Menu/References, which loads the form shown in Fig 4. Scroll through Available References until you locate the DirectX8 for *Visual Basic* Type Library and check the box. When you press "OK," the library is registered.

Define Variables, Buffers and DirectX Objects

Name the form in the Sound project frmSound. In the General section of frmSound, you will need to declare all of the variables, buffers and DirectX objects that will be used in the driver interface. Fig 5 provides the code that is to be copied into the General section. All definitions are commented in the code and should be self-explanatory when viewed in conjunction with the subroutine code.

Create the Audio Devices

We are now ready to create the DirectSound objects and set up the format of the capture and play buffers. Refer to the source code in Fig 6 during the following discussion.

The first step is to create the DirectSound and DirectSoundCapture

```
'Set up the DirectSound Objects and the Capture and Play Buffers
Sub CreateDevices()

    On Local Error Resume Next

    Set ds = dx.DirectSoundCreate(vbNullString)      'DirectSound object
    Set dsc = dx.DirectSoundCaptureCreate(vbNullString)    'DirectSound Capture

    'Check to se if Sound Card is properly installed
    If Err.Number <> 0 Then
        MsgBox "Unable to start DirectSound. Check proper sound card installation"
       End
    End If

    'Set the cooperative level to allow the Primary Buffer format to be set
    ds.SetCooperativeLevel Me.hWnd, DSSCL_PRIORITY

    'Set up format for capture buffer
    With dscbd
        With .fxFormat
            .nFormatTag = WAVE_FORMAT_PCM
            .nChannels = 2                    'Stereo
            .lSamplesPerSec = Fs              'Sampling rate in Hz
            .nBitsPerSample = 16              '16 bit samples
            .nBlockAlign = .nBitsPerSample / 8 * .nChannels
            .lAvgBytesPerSec = .lSamplesPerSec * .nBlockAlign
        End With
        .lFlags = DSCBCAPS_DEFAULT
        .lBufferBytes = (dscbd.fxFormat.nBlockAlign * CAPTURESIZE) 'Buffer Size
        CaptureBytes = .lBufferBytes \ 2      'Bytes for 1/2 of capture buffer
    End With

    Set dscb = dsc.CreateCaptureBuffer(dscbd)       'Create the capture buffer

    ' Set up format for secondary playback buffer
    With dsbd
        .fxFormat = dscbd.fxFormat
        .lBufferBytes = dscbd.lBufferBytes * 2      'Play is 2X Capture Buffer Size
        .lFlags = DSBCAPS_GLOBALFOCUS Or DSBCAPS_GETCURRENTPOSITION2
    End With

    dspbd = dsbd.fxFormat                       'Set Primary Buffer format
    dspb.SetFormat dspbd                        'to same as Secondary Buffer

    Set dsb = ds.CreateSoundBuffer(dsbd)       'Create the secondary buffer

End Sub
```

Fig 6—Create the DirectX capture and playback devices.

objects. We then check for an error to see if we have a compatible sound card installed. If not, an error message would be displayed to the user. Next, we set the cooperative level DSSCL_ PRIORITY to allow the Primary Buffer format to be set to the same as that of the Secondary Buffer. The code that follows sets up the DirectSoundCaptureBufferDescription format and creates the DirectSoundCaptureBuffer object. The format is set to 16-bit stereo at the sampling rate set by the constant *F*s.

Next, the DirectSoundBufferDescription is set to the same format as the DirectSoundCaptureBufferDescription. We then set the Primary Buffer format to that of the Secondary Buffer before creating the DirectSoundBuffer object.

Set the DirectX Events

As discussed earlier, the DirectSoundCaptureBuffer is divided into two blocks so that we can read from one block while capturing to the other. To do so, we must know when

```
'Set events for capture buffer notification at 0 and 1/2
Sub SetEvents()

    hEvent(0) = dx.CreateEvent(Me)          'Event handle for first half of buffer
    hEvent(1) = dx.CreateEvent(Me)          'Event handle for second half of buffer

    'Buffer Event 0 sets Write at 50% of buffer
    EVNT(0).hEventNotify = hEvent(0)
    EVNT(0).lOffset = (dscbd.lBufferBytes \ 2) - 1 'Set event to first half of capture buffer

    'Buffer Event 1 Write at 100% of buffer
    EVNT(1).hEventNotify = hEvent(1)
    EVNT(1).lOffset = dscbd.lBufferBytes - 1       'Set Event to second half of capture buffer

    dscb.SetNotificationPositions 2, EVNT()        'Set number of notification positions to 2

End Sub
```

Fig 7—Create the DirectX events.

```
'Create Devices and Set the DirectX8Events
Private Sub Form_Load()
    CreateDevices                       'Create DirectSound devices
    SetEvents                           'Set up DirectX events
End Sub

'Shut everything down and close application
Private Sub Form_Unload(Cancel As Integer)

    If Receiving = True Then
        dsb.Stop                        'Stop Playback
        dscb.Stop                       'Stop Capture
    End If

    Dim i As Integer
    For i = 0 To UBound(hEvent)         'Kill DirectX Events
        DoEvents
        If hEvent(i) Then dx.DestroyEvent hEvent(i)
    Next

    Set dx = Nothing                    'Destroy DirectX objects
    Set ds = Nothing
    Set dsc = Nothing
    Set dsb = Nothing
    Set dscb = Nothing

    Unload Me

End Sub
```

Fig 8—Create and destroy the DirectSound Devices and events.

DirectX has finished writing to a block. This is accomplished using the DirectXEvent8. Fig 7 provides the code necessary to set up the two events that occur when the lWrite cursor has reached 50% and 100% of the DirectSoundCaptureBuffer.

We begin by creating the two event handles hEvent(0) and hEvent(1). The code that follows creates a handle for each of the respective events and sets them to trigger after each half of the DirectSoundCaptureBuffer is filled. Finally, we set the number of notification positions to two and pass the name of the EVNT() event handle array to DirectX.

The CreateDevices and SetEvents subroutines should be called from the Form_Load() subroutine. The Form_Unload subroutine must stop capture and playback and destroy all of the DirectX objects before shutting down. The code for loading and unloading is shown in Fig 8.

Starting and Stopping Capture/Playback

Fig 9 illustrates how to start and stop the DirectSoundCaptureBuffer. The dscb.Start DSCBSTART_ LOOPING command starts the DirectSoundCaptureBuffer in a continuous circular loop. When it fills the first half of the buffer, it triggers the DirectX Event8 subroutine so that the data can be read, processed and sent to the DirectSoundBuffer. Note that the DirectSoundBuffer has not yet been started since we will quadruple buffer the output to prevent processor loading from causing gaps in the output. The FirstPass flag tells the event to start filling the DirectSoundBuffer for the first time before starting the buffer looping.

Processing the Direct-XEvent8

Once we have started the DirectSoundCaptureBuffer looping, the completion of each block will cause the DirectX Event8 code in Fig 10 to be executed. As we have noted, the events will occur when 50% and 100% of the buffer has been filled with data. Since the buffer is circular, it will begin again at the 0 location when the buffer is full to start the cycle all over again. Given a sampling rate of 44,100 Hz and 2048 samples per capture block, the block rate is calculated to be 44,100/2048 = 21.53 blocks/s or one block every 46.4 ms. Since the quad buffer is filled before starting playback the total delay from input to output is 4×46.4 ms = 185.6 ms.

The DirectX Event8_DXCallback event passes the eventid as a variable. The case statement at the beginning of the code determines from the eventid, which half of the DirectSoundCaptureBuffer has just been filled. With that information, we can calculate the starting address for reading each block from the DirectSoundCaptureBuffer to the inBuffer() array with the dscb.ReadBuffer command. Next, we simply pass the inBuffer() to the external DSP subroutine, which returns the processed data in the outBuffer() array.

Then we calculate the StartAddr and EndAddr for the next write location in the DirectSoundBuffer. Before writing to the buffer, we first check to make sure that we are not writing between the lWrite and lPlay cursors, which will cause portions of the buffer to be overwritten that have already been committed to the output. This will result in noise and distortion in the audio output. If an error occurs, the FirstPass flag is set to true and the pointers are reset to zero so that we flush the DirectSoundBuffer and start over. This effectively performs an automatic reset when the processor is overloaded, typically because of graphics intensive applications running alongside the SDR application.

If there are no overwrite errors, we write the outBuffer() array that was returned from the DSP routine to the next StartAddr to EndAddr in the DirectSoundBuffer. Important note: In the sample code, the DSP subroutine call is commented out and the inBuffer() array is passed directly to the DirectSoundBuffer for testing of the code. When the FirstPass flag is set to True, we capture and write four data blocks before starting playback looping with the .SetCurrentPosition 0 and .Play DSBPLAY_LOOPING commands.

The subroutine calls to StartTimer and StopTimer allow the average computational time of the event loop to be displayed in the immediate window. This is useful in measuring the efficiency of the DSP subroutine code that is called from the event. In normal operation, these subroutine calls should be commented out.

Parsing the Stereo Buffer into I and Q Signals

One more step that is required to use the captured signal in the DSP subroutine is to separate or parse the left and right channel data into the *I* and *Q* signals, respectively. This can be accomplished using the code in Fig 11. In 16-bit stereo, the left and right channels are interleaved in the inBuffer() and outBuffer(). The code simply copies the alternating 16-bit integer values to the RealIn()), (same as *I*) and ImagIn(), (same as *Q*) buffers respectively. Now we are ready to perform the magic of digital signal processing that we will discuss in the next article of the series.

Testing the Driver

To test the driver, connect an audio generator—or any other audio device, such as a receiver—to the line input of the sound card. Be sure to mute line-in on the mixer control panel so that you will not hear the audio directly through the operating system. You can open the mixer by double clicking on the speaker icon in the lower right corner of your Windows screen. It is also accessible through the Control Panel.

Now run the Sound application and press the On button. You should hear the audio playing through the driver. It will be delayed about 185 ms from the incoming audio because of the quadruple buffering. You can turn the mute control on the line-in mixer on and off to test the delay. It should sound like an echo. If so, you know that everything is operating properly.

Coming Up Next

In the next article, we will discuss in detail the DSP code that provides

```
'Turn Capture/Playback On
Private Sub cmdOn_Click()
    dscb.Start DSCBSTART_LOOPING      'Start Capture Looping
    Receiving = True                  'Set flag to receive mode
    FirstPass = True                  'This is the first pass after
Start
    OutPtr = 0                        'Starts writing to first buffer
End Sub

'Turn Capture/Playback Off
Private Sub cmdOff_Click()
    Receiving = False                 'Reset Receiving flag
    FirstPass = False                 'Reset FirstPass flag
    dscb.Stop                         'Stop Capture Loop
    dsb.Stop                          'Stop Playback Loop
End Sub
```

Fig 9—Start and stop the capture/playback buffers.

```
'Process the Capture events, call DSP routines, and output to Secondary Play Buffer
Private Sub DirectXEvent8_DXCallback (ByVal eventid As Long)

    StartTimer                          'Save loop start time

    Select Case eventid                 'Determine which Capture Block is ready
        Case hEvent(0)
            InPtr = 0                   'First half of Capture Buffer
        Case hEvent(1)
            InPtr = 1                   'Second half of Capture Buffer
    End Select

    StartAddr = InPtr * CaptureBytes    'Capture buffer starting address

     'Read from DirectX circular Capture Buffer to inBuffer
     dscb.ReadBuffer StartAddr, CaptureBytes, inBuffer(0), DSCBLOCK_DEFAULT

     'DSP Modulation/Demodulation - NOTE: THIS IS WHERE THE DSP CODE IS CALLED
'     DSP inBuffer, outBuffer

     StartAddr = OutPtr * CaptureBytes  'Play buffer starting address
     EndAddr = OutPtr + CaptureBytes - 1 'Play buffer ending address

    With dsb                            'Reference DirectSoundBuffer

            .GetCurrentPosition PlyCurs 'Get current Play position

             'If true the write is overlapping the lWrite cursor due to processor loading
             If PlyCurs.lWrite >= StartAddr _
                 And PlyCurs.lWrite <= EndAddr Then
                     FirstPass = True   'Restart play buffer
                     OutPtr = 0
                     StartAddr = 0
             End If

             'If true the write is overlapping the lPlay cursor due to processor loading
             If PlyCurs.lPlay >= StartAddr _
                 And PlyCurs.lPlay <= EndAddr Then
                     FirstPass = True   'Restart play buffer
                     OutPtr = 0
                     StartAddr = 0
             End If

         'Write outBuffer to DirectX circular Secondary Buffer. NOTE: writing inBuffer causes
direct pass through. Replace
         'with outBuffer below to when using DSP subroutine for modulation/demodulation
         .WriteBuffer StartAddr, CaptureBytes, inBuffer(0), DSBLOCK_DEFAULT

        OutPtr = IIf(OutPtr >= 3, 0, OutPtr + 1)       'Counts 0 to 3

        If FirstPass = True Then        'On FirstPass wait 4 counts before starting
            Pass = Pass + 1             'the Secondary Play buffer looping at 0
            If Pass = 3 Then            'This puts the Play buffer three Capture cycles
                FirstPass = False       'after the current one
                Pass = 0                'Reset the Pass counter
                .SetCurrentPosition 0   'Set playback position to zero
                .Play DSBPLAY_LOOPING   'Start playback looping
            End If
        End If

    End With

    StopTimer                           'Display average loop time in immediate window

End Sub
```

Fig 10—Process the DirectXEvent8 event. Note that the example code passes the inBuffer() directly to the DirectSoundBuffer without processing. The DSP subroutine call has been commented out for this illustration so that the audio input to the sound card will be passed directly to the audio output with a 185 ms delay. Destroy objects and events on exit.

```
Erase RealIn, ImagIn

    For S = 0 To CAPTURESIZE - 1 Step 2        'Copy I to RealIn and Q to ImagIn
        RealIn(S \ 2) = inBuffer(S)
        ImagIn(S \ 2) = inBuffer(S + 1)
    Next S
```

Fig 11—Code for parsing the stereo inBuffer() into in-phase and quadrature signals. This code must be imbedded into the DSP subroutine.

modulation and demodulation of SSB signals. Included will be source code for implementing ultra-high-performance variable band-pass filtering in the frequency domain, offset baseband IF processing and digital AGC.

Notes

[1]G. Youngblood, AC5OG, "A Software-Defined Radio for the Masses: Part 1," *QEX*, July/Aug 2002, pp 13-21.

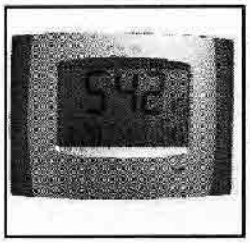
[2]Information on both DirectX and Windows Multimedia programming can be accessed on the Microsoft Developer Network (MSDN) Web site at **www.msdn. microsoft.com/library**. To download the DirectX Software Development Kit go to **msdn.microsoft.com/downloads/** and click on "Graphics and Multimedia" in the left-hand navigation window. Next click on "DirectX" and then "DirectX 8.1" (or a later version if available). The DirectX runtime driver may be downloaded from **www.microsoft.com/windows/directx/downloads/default.asp**.

[3]You can download this package from the ARRL Web **www.arrl.org/qexfiles/**. Look for 0902Youngblood.zip. ☐☐

ATOMIC TIME

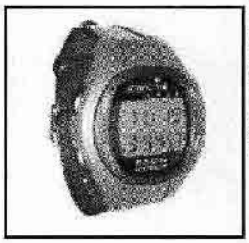
1010 Jorie Blvd. #332 Oak Brook, IL. 60523

Tell time by the U.S. Atomic Clock -The official U.S. time that governs ship movements, radio stations, space flights, and warplanes. With small radio receivers hidden inside our timepieces, they automatically syncronize to the U.S. Atomic Clock (which measures each second of time as 9,192,631,770 vibrations of a cesium 133 atom in a vacuum) and give time which is accurate to 1 second every million years. Our timepieces even account automatically for daylight saving time, leap years, and leap seconds. Accept only the best, most precise, and reliable timepieces in the world. Atomic Time.



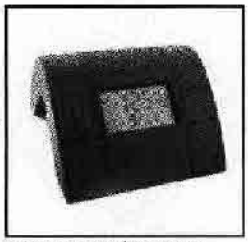
Atomic Time
0645R Digital
Light/Dark Grey
12 or 24 hr. format
day, date, month
temp. alarm
10.5" x 7.5"
$29.95

Junghans MEGA
Atomic Carbon
Carbon Fiber case
sapphire glass
crystal lens, black
leather band, LCD
with day & date.
$239.20



Atomic Time
W102 Digital
Hour, minute, second, day, date,
blue polymer case
backlight, alarm &
stopwatch
$49.95

Atomic Dual Time-
Dual Alarm Clock
TLWA201
Available in black
or silver, backlit,
displays local time
and any world time
$45.00



1-800-985-8463
www.atomictime.com
30 Day Money-Back Guarantee
Call for our FREE Brochure

All items add $7.95 S/H
IL. res. add 6.75% sales tax

We Design And Manufacture To Meet Your Requirements

*Protoype or Production Quantities

800-522-2253

This Number May Not Save Your Life...

But it could make it a lot easier! Especially when it comes to ordering non-standard connectors.

RF/MICROWAVE CONNECTORS, CABLES AND ASSEMBLIES

- Specials our specialty. Virtually any SMA, N, TNC, HN, LC, RP, BNC, SMB, or SMC delivered in 2-4 weeks.
- Cross reference library to all major manufacturers.
- Experts in supplying "hard to get" RF connectors.
- Our adapters can satisfy virtually any combination of requirements between series.
- Extensive inventory of passive RF/Microwave components including attenuators, terminations and dividers.
- No minimum order.

NEMAL
Cable & Connectors for the Electronics Industry

NEMAL ELECTRONICS INTERNATIONAL, INC.
12240 N.E. 14TH AVENUE
NORTH MIAMI, FL 33161
TEL: 305-899-0900 • FAX: 305-895-8178
E-MAIL: INFO@NEMAL.COM
BRASIL: (011) 5535-2368

URL: WWW.NEMAL.COM

QEX

American Radio Relay League
225 Main Street
Newington, CT 06111-1494 USA

For one year (6 bimonthly issues) of QEX:
In the US:
☐ ARRL Member $22.00
☐ Non-Member $34.00
In Canada, Mexico and U.S. by First Class mail:
☐ ARRL Member $35.00
☐ Non-Member $47.00
Elsewhere by Surface Mail (4-6 week delivery):
☐ ARRL Member $27.00
☐ Non-Member $39.00
Elsewhere by Airmail:
☐ ARRL Member $55.00
☐ Non-Member $67.00

QEX Subscription Order Card

QEX, the Forum for Communications Experimenters is available at the rates shown at left. Maximum term is 6 issues, and because of the uncertainty of postal rates, prices are subject to change without notice.

Subscribe toll-free with your credit card 1-888-277-5289

☐ Renewal ☐ New Subscription

Name_____ Call _____

Address_____

City _____ State or Province _____ Postal Code _____

Payment Enclosed ☐

Charge:
☐ MasterCard ☐ VISA ☐ Discover Card ☐ American Express

Account #_____ Good thru _____

Signature_____ Date_____

Remittance must be in US funds and checks must be drawn on a bank in the US.
Prices subject to change without notice.

11/98

A Software-Defined Radio for the Masses, Part 3

Learn how to use DSP to make the PC sound-card interface from Part 2 into a functional software-defined radio. We also explore a powerful filtering technique called FFT fast-convolution filtering.

By Gerald Youngblood, AC5OG

Part 1[1] of this series provided a general description of digital signal processing (DSP) as used in software-defined radios (SDRs) and included an overview of a full-featured radio that uses a PC to perform all DSP and control functions. Part 2[2] described *Visual Basic* source code that implements a full-duplex quadrature interface to a PC sound card.

As previously described, *in-phase (I)* and *quadrature (Q)* signals give the ability to modulate or demodulate virtually any type of signal. The Tayloe Detector, described in Part 1, is a simple method of converting a modulated RF signal to baseband in quadrature, so that it can be presented to the left and right inputs of a stereo PC sound card for signal processing. The full-duplex DirectX8 interface, described in Part 2, accomplishes the input and output of the sampled quadrature signals. The sound-card interface provides an input buffer array, *inBuffer*(), and an output buffer array, *outBuffer*(), through which the

[1]Notes appear on page 36.

8900 Marybank Dr
Austin, TX 78750
AC5OG@arrl.net

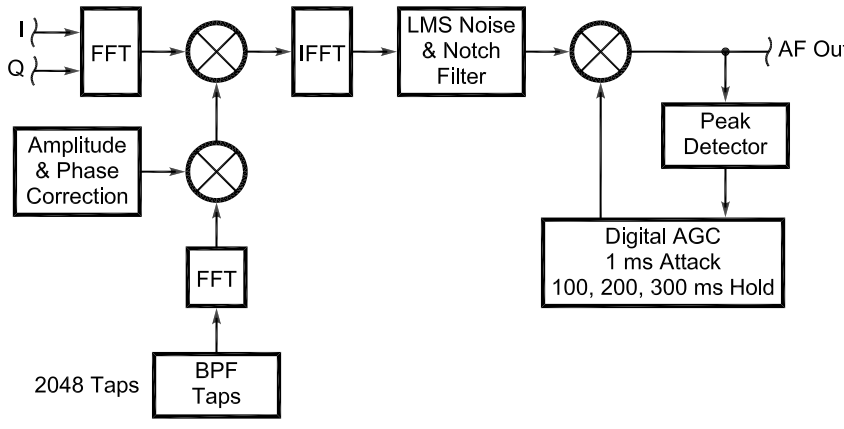


Fig 1—DSP software architecture block diagram.

DSP code receives the captured signal and then outputs the processed signal data.

This article extends the sound-card interface to a functional SDR receiver demonstration. To accomplish this, the following functions are implemented in software:

- Split the stereo sound buffers into *I* and *Q* channels.
- Conversion from the time domain into the frequency domain using a *fast Fourier transform (*FFT*).*
- Cartesian-to-polar conversion of the signal vectors.
- Frequency translation from the 11.25 kHz-offset baseband IF to 0 Hz.
- Sideband selection.
- Band-pass filter coefficient generation.
- FFT fast-convolution filtering.
- Conversion back to the time domain with an inverse fast Fourier transform (IFFT).
- Digital automatic gain control (AGC) with variable hang time.
- Transfer of the processed signal to the output buffer for transmit or receive operation.

The demonstration source code may

```
Public Const Fs As Long = 44100                          'Sampling frequency in samples per
                                                         'second
Public Const NFFT As Long = 4096                         'Number of FFT bins
Public Const BLKSIZE As Long = 2048                      'Number of samples in capture/play block
Public Const CAPTURESIZE As Long = 4096                  'Number of samples in Capture Buffer
Public Const FILTERTAPS As Long = 2048                   'Number of taps in bandpass filter
Private BinSize As Single                                'Size of FFT Bins in Hz

Private order As Long                                    'Calculate Order power of 2 from NFFT
Private filterM(NFFT) As Double                          'Polar Magnitude of filter freq resp
Private filterP(NFFT) As Double                          'Polar Phase of filter freq resp
Private RealIn(NFFT) As Double                           'FFT buffers
Private RealOut(NFFT) As Double
Private ImagIn(NFFT) As Double
Private ImagOut(NFFT) As Double

Private IOverlap(NFFT - FILTERTAPS - 1) As Double         'Overlap prev FFT/IFFT
Private QOverlap(NFFT - FILTERTAPS - 1) As Double         'Overlap prev FFT/IFFT

Private RealOut_1(NFFT) As Double                        'Fast Convolution Filter buffers
Private RealOut_2(NFFT) As Double
Private ImagOut_1(NFFT) As Double
Private ImagOut_2(NFFT) As Double

Public FHigh As Long                                     'High frequency cutoff in Hz
Public FLow As Long                                      'Low frequency cutoff in Hz
Public Fl As Double                                      'Low frequency cutoff as fraction of Fs
Public Fh As Double                                      'High frequency cutoff as fraction of Fs
Public SSB As Boolean                                    'True for Single Sideband Modes
Public USB As Boolean                                    'Sideband select variable
Public TX As Boolean                                     'Transmit mode selected
Public IFShift As Boolean                                'True for 11.025KHz IF

Public AGC As Boolean                                    'AGC enabled
Public AGCHang As Long                                   'AGC AGCHang time factor
Public AGCMode As Long                                   'Saves the AGC Mode selection
Public RXHang As Long                                    'Save RX Hang time setting
Public AGCLoop As Long                                   'AGC AGCHang time buffer counter
Private Vpk As Double                                    'Peak filtered output signal
Private G(24) As Double                                  'Gain AGCHang time buffer
Private Gain As Double                                   'Gain state setting for AGC
Private PrevGain As Double                               'AGC Gain during previous input block
Private GainStep As Double                               'AGC attack time steps
Private GainDB As Double                                 'AGC Gain in dB
Private TempOut(BLKSIZE) As Double                       'Temp buffer to compute Gain
Public MaxGain As Long                                   'Maximum AGC Gain factor

Private FFTBins As Long                                  'Number of FFT Bins for Display
Private M(NFFT) As Double                                'Double precision polar magnitude
Private P(NFFT) As Double                                'Double precision phase angle
Private S As Long                                        'Loop counter for samples
```

Fig 2—Variable declarations.

be downloaded from *ARRLWeb*.[3] The software requires the *dynamic link library (DLL)* files from the Intel Signal Processing Library[4] to be located in the working directory. These files are included with the demo software.

The Software Architecture

Fig 1 provides a block diagram of the DSP software architecture. The architecture works equally well for both transmit and receive with only a few lines of code changing between the two. While the block diagram illustrates functional modules for *Amplitude and Phase Correction* and the *LMS Noise and Notch Filter*, discussion of these features is beyond the scope of this article.

Amplitude and phase correction permits imperfections in phase and amplitude imbalance created in the analog circuitry to be corrected in the frequency domain. LMS noise and notch filters[5] are an adaptive form of *finite impulse response* (FIR) filtering that accomplishes noise reduction in the time domain. There are other techniques for noise reduction that can be accomplished in the frequency domain such as *spectral subtraction,*[6] *correlation*[7] and *FFT averaging.*[8]

Parse the Input Buffers to Get *I* and *Q* Signal Vectors

Fig 2 provides the variable and constant declarations for the demonstration code. The code for parsing the *inBuffer*() is illustrated in Fig 3. The left and right signal inputs must be parsed into *I* and *Q* signal channels before they are presented to the FFT input. The 16-bit integer left- and right-channel samples are interleaved, therefore the code shown in Fig 3 must be used to split the signals. The arrays *RealIn*() and *RealOut*() are used to store the *I* signal vectors and the arrays *ImagIn*() and *ImagOut*() are used to store the *Q* signal vectors. This corresponds to the nomenclature used in the complex FFT algorithm. It is not critical which of the *I* and *Q* channels goes to which input because one can simply reverse the code in Fig 3 if the sidebands are inverted.

The FFT: Conversion to the Frequency Domain

Part 1 of this series discussed how the FFT is used to convert discrete-time sampled signals from the time domain into the frequency domain (see Note 1). The FFT is quite complex to derive mathematically and somewhat tedious to code. Fortunately, Intel has provided performance-optimized code in DLL form that can be called from a single line of code for this and other important DSP functions (see Note 4).

The FFT effectively consists of a series of very narrow band-pass filters, the outputs of which are called *bins,* as illustrated in Fig 4. Each bin has a magnitude and phase value representative of the sampled input signal's content at the respective bin's center frequency. Overlap of adjacent bins resembles the output of a *comb filter* as discussed in Part 1.

The PC SDR uses a 4096-bin FFT. With a sampling rate of 44,100 Hz, the bandwidth of each bin is 10.7666 Hz (44,100/4096), and the center frequency of each bin is the bin number times the bandwidth. Notice in Fig 4 that with respect to the center frequency of the sampled quadrature signal, the upper sideband is located in bins 1 through 2047, and the lower sideband is located in bins 2048 through 4095. Bin 0 contains the carrier translated to 0 Hz. An FFT performed on an analytic signal $I + jQ$ allows positive and negative frequencies to be analyzed separately.

The Turtle Beach Santa Cruz sound card I use has a 3-dB frequency response of approximately 10 Hz to 20 kHz. (Note: the data sheet states a high-frequency cutoff of 120 kHz, which has to be a typographical error, given the 48-kHz maximum sampling rate). Since we sample the RF signal in quadrature, the sampling rate is effectively doubled (44,100 Hz times

```
Erase RealIn, ImagIn

For S = 0 To CAPTURESIZE - 1 Step 2      'Copy I to RealIn and Q to ImagIn
     RealIn(S \ 2) = inBuffer(S + 1)     'Zero stuffing second half of
     ImagIn(S \ 2) = inBuffer(S)         'RealIn and ImagIn Next S
```

Fig 3—Parsing input buffers into *I* and *Q* signal vectors.

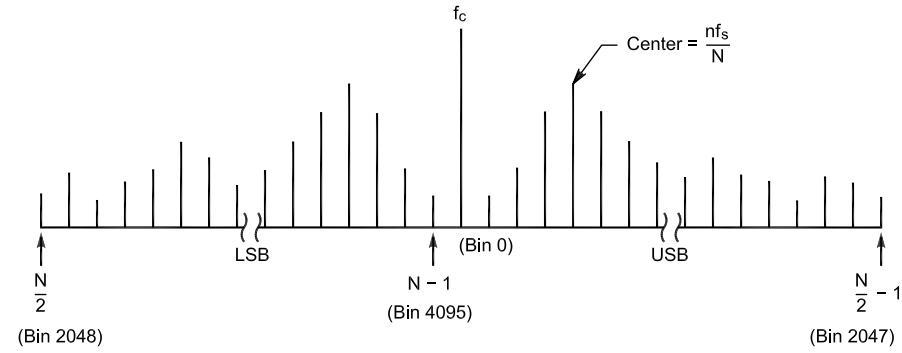


Fig 4—FFT output bins.

```
nspzrFftNip RealIn, ImagIn, RealOut, ImagOut, order, NSP_Forw
nspdbrCartToPolar RealOut, ImagOut, M, P, NFFT   'Cartesian to polar
```

Fig 5—Time domain to frequency domain conversion using the FFT.

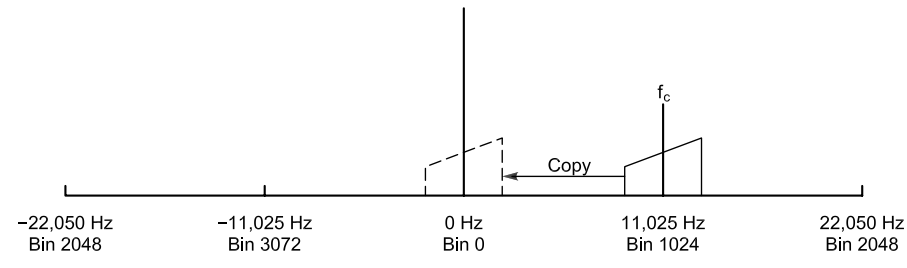


Fig 6—Offset baseband IF diagram. The local oscillator is shifted by 11.025 kHz so that the desired-signal carrier frequency is centered at an 11,025-Hz offset within the FFT output. To shift the signal for subsequent filtering the desired bins are simply copied to center the carrier frequency, f_c, at 0 Hz.

two channels yields an 88,200-Hz effective sampling rate). This means that the output spectrum of the FFT will be twice that of a single sampled channel. In our case, the total output bandwidth of the FFT will be 10.7666 Hz times 4096 or 44,100 Hz. Since most sound cards roll off near 20 kHz, we are probably limited to a total bandwidth of approximately 40 kHz.

Fig 5 shows the DLL calls to the Intel library for the FFT and subsequent conversion of the signal vectors from the Cartesian coordinate system to the Polar coordinate system. The nspzrFftNip routine takes the time domain *RealIn*() and *ImagIn*() vectors and converts them into frequency domain *RealOut*() and *ImagOut*() vectors. The order of the FFT is computed in the routine that calculates the filter coefficients as will be discussed later. NSP_Forw is a constant that tells the routine to perform the forward FFT conversion.

In the Cartesian system the signal is represented by the magnitudes of two vectors, one on the *Real* or *x* plane and one on the *Imaginary* or *y* plane. These vectors may be converted to a single vector with a magnitude (*M*) and a phase angle (*P*) in the polar system. Depending on the specific DSP algorithm we wish to perform, one coordinate system or the other may be more efficient. I use the polar coordinate system for most of the signal processing in this example. The nspdbrCartToPolar routine converts the output of the FFT to a polar vector consisting of the magnitudes in *M*() and the phase values in *P*(). This function simultaneously performs Eqs 3 and 4 in Part 1 of this article series.

Offset Baseband IF Conversion to Zero Hertz

My original software centered the RF carrier frequency at bin 0 (0 Hz). With this implementation, one can display (and hear) the entire 44-kHz spectrum in real time. One of the problems encountered with direct-conversion or zero-IF receivers is that noise increases substantially near 0 Hz. This is caused by several mechanisms: 1/*f* noise in the active components, 60/120-Hz noise from the ac power lines, microphonic noise caused by mechanical vibration and local-oscillator phase noise. This can be a problem for weak-signal work because most people tune CW signals for a 700-1000 Hz tone. Fortunately, much of this noise disappears above 1 kHz.

Given that we have 44 kHz of spectrum to work with, we can offset the digital IF to any frequency within the FFT output range. It is simply a matter of deciding which FFT bin to designate as the carrier frequency and then offsetting the local oscillator by the appropriate amount. We then copy the respective bins for the desired sideband so that they are located at 0 Hz for subsequent processing. In the PC SDR, I have chosen to use an offset IF of 11,025 Hz, which is one fourth of the sampling rate, as shown in Fig 6.

Fig 7 provides the source code for shifting the offset IF to 0 Hz. The carrier frequency of 11,025 Hz is shifted to bin 0 and the upper sideband is shifted to bins 1 through 1023. The lower sideband is shifted to bins 3072 to 4094. The code allows the IF shift to be enabled or disabled, as is required for transmitting.

Selecting the Sideband

So how do we select sideband? We store zeros in the bins we don't want to hear. How simple is that? If it were possible to have perfect analog amplitude and phase balance on the sampled *I* and *Q* input signals, we would have infinite sideband suppression. Since that is not possible, any imbalance will show up as an image in the passband of the receiver. Fortunately, these imbalances can be cor-

```
IFShift = True                          'Force to True for the demo

    If IFShift = True Then              'Shift sidebands from 11.025KHz IF
        For S = 0 To 1023
            If USB Then
                M(S) = M(S + 1024)      'Move upper sideband to 0Hz
                P(S) = P(S + 1024)
            Else
                M(S + 3072) = M(S + 1)  'Move lower sideband to 0Hz
                P(S + 3072) = P(S + 1)
            End If
        Next
    End If
```

Fig 7—Code for down conversion from offset baseband IF to 0 Hz.

```
If SSB = True Then                          'SSB or CW Modes
        If USB = True Then
            For S = FFTBins To NFFT - 1     'Zero out lower sideband
                M(S) = 0
            Next
        Else
            For S = 0 To FFTBins - 1        'Zero out upper sideband
                M(S) = 0
            Next
        End If
    End If
```

Fig 8—Sideband selection code.

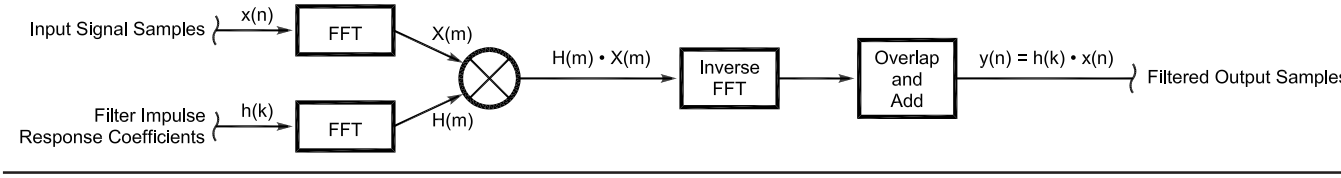


Fig 9—FFT fast-convolution-filtering block diagram. The filter impulse-response coefficients are first converted to the frequency domain using the FFT and stored for repeated use by the filter routine. Each signal block is transformed by the FFT and subsequently multiplied by the filter frequency-response magnitudes. The resulting filtered signal is transformed back into the time domain using the inverse FFT. The Overlap/Add routine corrects the signal for circular convolution.

rected through DSP code either in the time domain before the FFT or in the frequency domain after the FFT. These techniques are beyond the scope of this discussion, but I may cover them in a future article. My prototype using INA103 instrumentation amplifiers achieves approximately 40 dB of opposite sideband rejection *without* correction in software.

The code for zeroing the opposite sideband is provided in Fig 8. The lower sideband is located in the high-numbered bins and the upper sideband is located in the low-numbered bins. To save time, I only zero the number of bins contained in the *FFTBins* variable.

FFT Fast-Convolution Filtering Magic

Every DSP text I have read on single-sideband modulation and demodulation describes the IF sampling approach. In this method, the A/D converter samples the signal at an IF such as 40 kHz. The signal is then quadrature down-converted in software to baseband and filtered using finite impulse response (FIR)[9] filters. Such a system was described in Doug Smith's *QEX* article called, "Signals, Samples, and Stuff: A DSP Tutorial (Part 1)."[10] With this approach, all processing is done in the time domain.

For the PC SDR, I chose to use a very different approach called *FFT fast-convolution filtering* (also called *FFT convolution*) that performs all filtering functions in the frequency domain.[11] An FIR filter performs convolution of an input signal with a filter impulse response in the time domain. Convolution is the mathematical means of combining two signals (for example, an input signal and a filter impulse response) to form a third signal (the filtered output signal).[12] The time-domain approach works very well for a small number of filter taps. What if we want to build a very-high-performance filter with 1024 or more taps? The processing overhead of the FIR filter may become prohibitive. It turns out that an important property of the Fourier transform is that convolution in the time domain is equal to multiplication in the frequency domain. Instead of directly convolving the input signal with the windowed filter impulse response, as with a FIR filter, we take the respective FFTs of the input signal and the filter impulse response and simply multiply them together, as shown in Fig 9. To get back to the time domain, we perform the inverse FFT of the product. FFT convolution is often faster than direct convolution for filter kernels longer than 64 taps, and it produces exactly the same result.

For me, FFT convolution is easier to understand than direct convolution because I mentally visualize filters in the frequency domain. As described in Part 1 of this series, the output of the complex FFT may be thought of as a long bank of narrow band-pass filters aligned around the carrier frequency (bin 0), as shown in Fig 4. Fig 10 illustrates the process of FFT convolution of a transformed filter impulse response with a transformed input signal. Once the signal is transformed back to the time domain by the inverse FFT, we must then perform a process called the *overlap/add method*. This is because the process of convolution produces an output signal that is

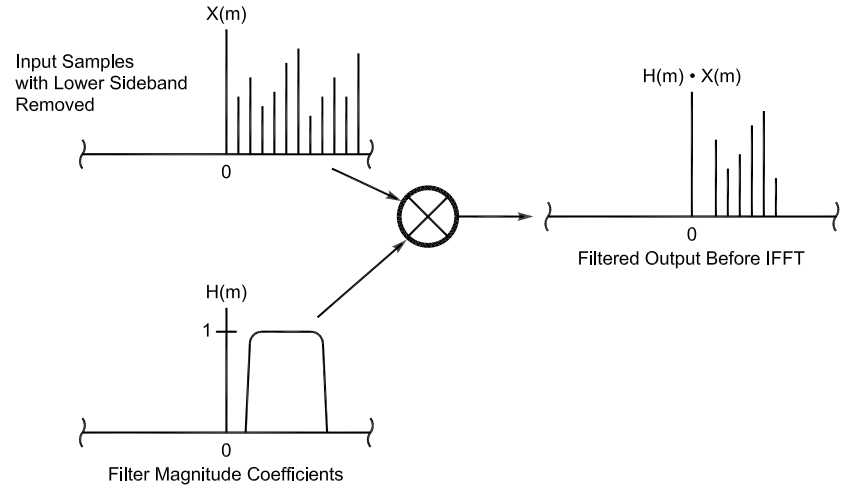


Fig 10—FFT fast convolution filtering output. When the filter-magnitude coefficients are multiplied by the signal-bin values, the resulting output bins contain values only within the pass-band of the filter.

```
Public Static Sub CalcFilter(FLow As Long, FHigh As Long)
Static Rh(NFFT) As Double           'Impulse response for bandpass filter
Static Ih(NFFT) As Double           'Imaginary set to zero
Static reH(NFFT) As Double          'Real part of filter response
Static imH(NFFT) As Double          'Imaginary part of filter response

Erase Ih

Fh = FHigh / Fs                     'Compute high and low cutoff
Fl = FLow / Fs                      'as a fraction of Fs
BinSize = Fs / NFFT                 'Compute FFT Bin size in Hz

FFTBins = (FHigh / BinSize) + 50    'Number of FFT Bins in filter width

order = NFFT                        'Compute order as NFFT power of 2
Dim O As Long

For O = 1 To 16                     'Calculate the filter order
     order = order \ 2
     If order = 1 Then
         order = O
         Exit For
     End If
Next

'Calculate infinite impulse response bandpass filter coefficients
'with window
nspdFirBandpass Fl, Fh, Rh, FILTERTAPS, NSP_WinBlackmanOpt, 1

'Compute the complex frequency domain of the bandpass filter
nspzrFftNip Rh, Ih, reH, imH, order, NSP_Forw
nspdbrCartToPolar reH, imH, filterM, filterP, NFFT

End Sub
```

Fig 11—Code for the generating bandpass filter coefficients in the frequency domain.

equal in length to the sum of the input samples plus the filter taps minus one. I will not attempt to explain the concept here because it is best described in the references.[13]

Fig 11 provides the source code for producing the frequency-domain band-pass filter coefficients. The CalcFilter subroutine is passed the low-frequency cutoff, *FLow*, and the high-frequency cutoff, *FHigh*, for the filter response. The cutoff frequencies are then converted to their respective fractions of the sampling rate for use by the filter-generation routine, nspdFirBandpass. The FFT order is also determined in this subroutine, based on the size of the FFT, NFFT. The nspdFirBandpass computes the impulse response of the band-pass filter of bandwidth *Fl*() to *Fh*() and a length of *FILTERTAPS*. It then places the result in the array variable *Rh*(). The NSP_WinBlackmanOpt causes the impulse response to be windowed by a Blackman window function. For a discussion of windowing, refer to the DSP Guide.[14] The value of "1" that is passed to the routine causes the result to be normalized.

Next, the impulse response is converted to the frequency domain by nspzrFftNip. The input parameters are *Rh*(), the real part of the impulse response, and *Ih*(), the imaginary part that has been set to zero. NSP_Forw tells the routine to perform the forward FFT. We next convert the frequency-domain result of the FFT, *reH*() and *imH*(), to polar form using the nspdbrCartToPolar routine. The filter magnitudes, *filterM*(), and filter phase, *filterP*(), are stored for use in the FFT fast convolution filter. Other than when we manually change the band-pass filter selection, the filter response does not change. This means that we only have to calculate the filter response once when the filter is first selected by the user.

Fig 12 provides the code for an FFT fast-convolution filter. Using the nspdbMpy2 routine, the signal-spectrum magnitude bins, *M*(), are multiplied by the filter frequency-response magnitude bins, *filterM*(), to generate the resulting in-place filtered magnitude-response bins, *M*(). We then use nspdbAdd2 to add the signal phase bins, *P*(), to the filter phase bins, *filterP*(), with the result stored in-place in the filtered phase-response bins, *P*(). Notice that FFT convolution can also be performed in Cartesian coordinates using the method shown in Fig 13, although this method requires more computational resources. Other uses of the frequency-domain magnitude values include FFT averaging, digital squelch and spectrum display.

Fig 14 shows the actual spectral output of a 500-Hz filter using wide-bandwidth noise input and FFT averaging of the signal over several seconds. This provides a good picture of the frequency response and shape of

```
nspdbMpy2 filterM, M, NFFT        'Multiply Magnitude Bins

nspdbAdd2 filterP, P, NFFT        'Add Phase Bins
```

Fig 12—FFT fast convolution filtering code using polar vectors.

```
'Compute: RealIn(s) = (RealOut(s) * reH(s)) - (ImagOut(s) * imH(s))
    nspdbMpy3 RealOut, reH, RealOut_1, NFFT
    nspdbMpy3 ImagOut, imH, ImagOut_1, NFFT
    nspdbSub3 RealOut_1, ImagOut_1, RealIn, NFFT  'RealIn for IFFT

    'Compute: ImagIn(s) = (RealOut(s) * imH(s)) + (ImagOut(s) * reH(s))
    nspdbMpy3 RealOut, imH, RealOut_2, NFFT
    nspdbMpy3 ImagOut, reH, ImagOut_2, NFFT
    nspdbAdd3 RealOut_2, ImagOut_2, ImagIn, NFFT  'ImagIn for IFFT
```

Fig 13—Alternate FFT fast convolution filtering code using cartesian vectors.

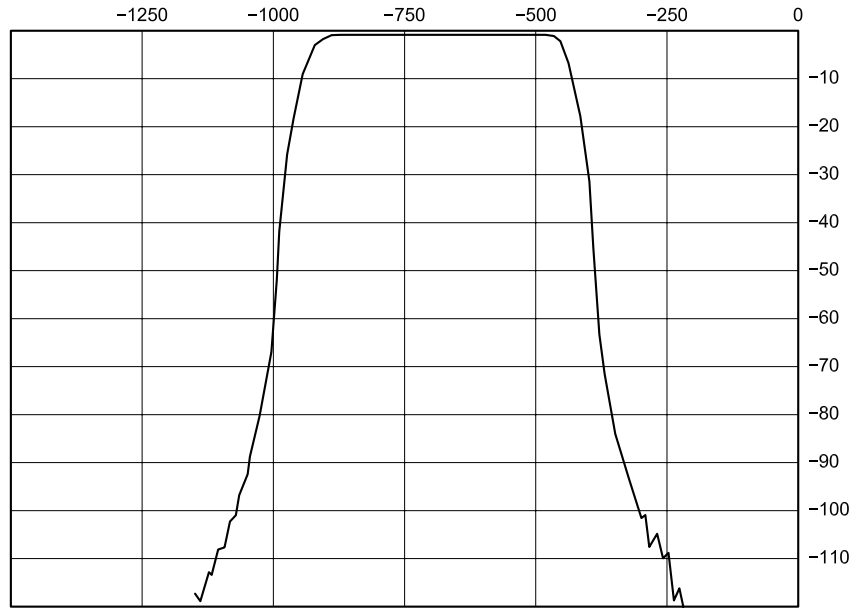


Fig 14—Actual 500-Hz CW filter pass-band display. FFT fast-convolution filtering is used with 2048 filter taps to produce a 1.05 shape factor from 3 dB to 60 dB down and over 120 dB of stop-band attenuation just 250 Hz beyond the 3 dB points.

```
'Convert polar to cartesian
   nspdbrPolarToCart M, P, RealIn, ImagIn, NFFT

   'Inverse FFT to convert back to time domain
   nspzrFftNip RealIn, ImagIn, RealOut, ImagOut, order, NSP_Inv

   'Overlap and Add from last FFT/IFFT:  RealOut(s) = RealOut(s) + Overlap(s)
   nspdbAdd3 RealOut, IOverlap, RealOut, FILTERTAPS - 2
   nspdbAdd3 ImagOut, QOverlap, ImagOut, FILTERTAPS - 2

   'Save Overlap for next pass
   For S = BLKSIZE To NFFT - 1
       IOverlap(S - BLKSIZE) = RealOut(S)
       QOverlap(S - BLKSIZE) = ImagOut(S)
  Next
```

Fig 15—Inverse FFT and overlap/add code.

the filter. The shape factor of the 2048-tap filter is 1.05 from the 3-dB to the 60-dB points (most manufacturers measure from 6 dB to 60 dB, a more lenient specification). Notice that the stop-band attenuation is greater than 120 dB at roughly 250 Hz from the 3-dB points. This is truly a brick-wall filter!

An interesting fact about this method is that the window is applied to the filter impulse response rather than the input signal. The filter response is normalized so signals within the passband are not attenuated in the frequency domain. I believe that this normalization of the filter response removes the usual attenuation associated with windowing the signal before performing the FFT. To overcome such windowing attenuation, it is typical to apply a 50-75% overlap in the time-domain sampling process and average the FFTs in the frequency domain. I would appreciate comments from knowledgeable readers on this hypothesis.

The IFFT and Overlap/Add—Conversion Back to the Time Domain

Before returning to the time domain, we must first convert back to Cartesian coordinates by using nspdbrPolarToCart as illustrated in Fig 15. Then by setting the NSP_Inv flag, the inverse FFT is performed by nspzrFftNip, which places the time-domain outputs in *RealOut*() and *ImagOut*(), respectively. As discussed previously, we must now overlap and add a portion of the signal from the previous capture cycle as described in the DSP Guide (see Note 13). *Ioverlap*() and *Qoverlap*() store the in-phase and quadrature overlap signals from the last pass to be added to the new signal block using the nspdbAdd3 routine.

Digital AGC with Variable Hang Time

The digital AGC code in Fig 16 provides fast-attack and -decay gain

```
If AGC = True Then

        'If true increment AGCLoop counter, otherwise reset to zero
        AGCLoop = IIf(AGCLoop < AGCHang - 1, AGCLoop + 1, 0)

        nspdbrCartToPolar RealOut, ImagOut, M, P, BLKSIZE  'Envelope Polar Magnitude

        Vpk = nspdMax(M, BLKSIZE)                          'Get peak magnitude

        If Vpk <> 0 Then                                   'Check for divide by zero
            G(AGCLoop) = 16384 / Vpk                       'AGC gain factor with 6 dB headroom
            Gain = nspdMin(G, AGCHang)                     'Find peak gain reduction (Min)
        End If

        If Gain > MaxGain Then Gain = MaxGain              'Limit Gain to MaxGain

        If Gain < PrevGain Then                            'AGC Gain is decreasing
            GainStep = (PrevGain - Gain) / 44              '44 Sample ramp = 1 ms attack time
            For S = 0 To 43                                'Ramp Gain down over 1 ms period
                M(S) = M(S) * (PrevGain - ((S + 1) * GainStep))
            Next
            For S = 44 To BLKSIZE - 1                      'Multiply remaining Envelope by Gain
                M(S) = M(S) * Gain
            Next
        Else
            If Gain > PrevGain Then                        'AGC Gain is increasing
                GainStep = (Gain - PrevGain) / 44          '44 Sample ramp = 1 ms decay time
                For S = 0 To 43                            'Ramp Gain up over 1 ms period
                    M(S) = M(S) * (PrevGain + ((S + 1) * GainStep))
                Next
                For S = 44 To BLKSIZE - 1                  'Multiply remaining Envelope by Gain
                    M(S) = M(S) * Gain
                Next
            Else
                nspdbMpy1 Gain, M, BLKSIZE                 'Multiply Envelope by AGC gain
            End If
        End If

        PrevGain = Gain                                    'Save Gain for next loop

        nspdbThresh1 M, BLKSIZE, 32760, NSP_GT             'Hard limiter to prevent overflow

    End If
```

Fig 16 – Digital AGC code.

control with variable hang time. Both attack and decay occur in approximately 1 ms, but the hang time may be set to any desired value in increments of 46 ms. I have chosen to implement the attack/decay with a linear ramp function rather than an exponential function as described in DSP communications texts.[15] It works extremely well and is intuitive to code. The flow diagram in Fig 17 outlines the logic used in the AGC algorithm.

Refer to Figs 16 and 17 for the following description. First, we check to see if the AGC is turned on. If so, we increment *AGCLoop*, the counter for AGC hang-time loops. Each pass through the code is equal to a hang time of 46 ms. PC SDR provides hang-time loop settings of 3 (fast, 132 ms), 5 (medium, 230 ms), 7 (slow, 322 ms) and 22 (long, 1.01 s). The hang-time setting is stored in the *AGCHang* variable. Once the hang-time counter resets, the decay occurs on a 1-ms linear slope.

To determine the AGC gain requirement, we must detect the envelope of the demodulated signal. This is easily accomplished by converting from Cartesian to polar coordinates. The value of *M*() is the envelope, or magnitude, of the signal. The phase vector can be ignored insofar as AGC is concerned. We will need to save the phase values, though, for conversion back to Cartesian coordinates later. Once we have the magnitudes stored in *M*(), it is a simple matter to find the peak magnitude and store it in *Vpk* with the function nspdMax. After checking to prevent a divide-by-zero error, we compute a gain factor relative to 50% of the full-scale value. This provides 6 dB of headroom from the signal peak to the full-scale output value of the DAC. On each pass, the gain factor is stored in the *G*() array so that we can find the peak gain reduction during the hang-time period using the nspdMin function. The peak gain-reduction factor is then stored in the *Gain* variable. Note that *Gain* is saved as a ratio and not in decibels, so that no log/antilog conversion is needed.

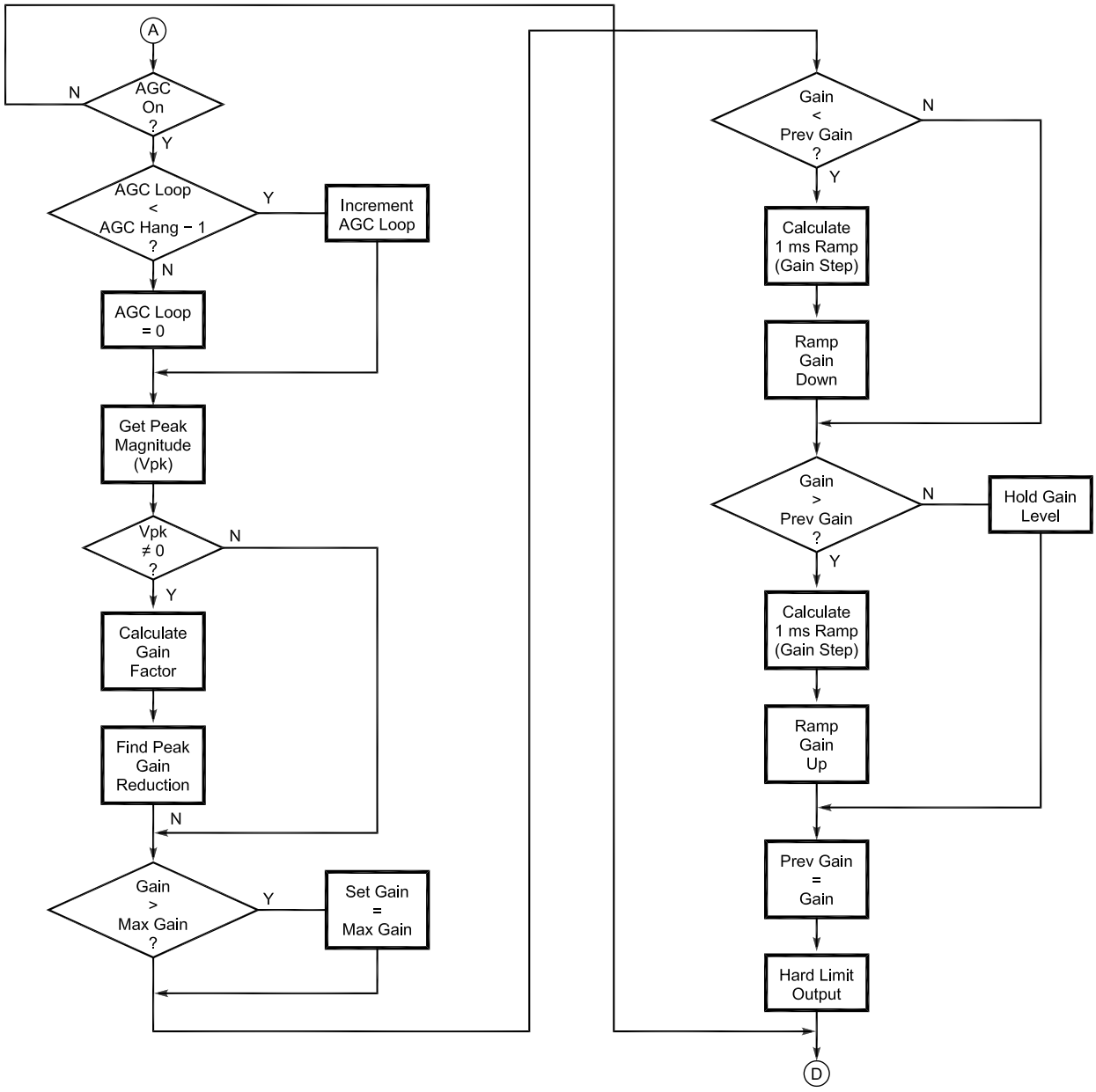


Fig 17—Digital AGC flow diagram.

The next step is to limit *Gain* to the *MaxGain* value, which may be set by the user. This system functions much like an IF-gain control allowing *Gain* to vary from negative values up to the *MaxGain* setting. Although not provided in the example code, it is a simple task to create a front panel control in *Visual Basic* to manually set the *MaxGain* value.

Next, we determine if the gain must be increased, decreased or left unchanged. If *Gain* is less than *PrevGain* (that is the *Gain* setting from the signal block stored on the last pass through the code), we ramp the gain down linearly over 44 samples. This yields an attack time of approximately 1 ms at a 44,100-Hz sampling rate. *GainStep* is the slope of the ramp per sample time calculated from the *PrevGain* and *Gain* values. We then incrementally ramp down the first 44 samples by the *GainStep* value. Once ramped to the new *Gain* value, we multiply the remaining samples by the fixed *Gain* value.

If *Gain* is increasing from the *PrevGain* value, the process is simply reversed. If *Gain* has not changed, all samples are multiplied by the current *Gain* setting. After the signal block has been processed, *Gain* is saved in *PrevGain* for the next signal block. Finally, nspdbThresh1 implements a hard limiter at roughly the maximum output level of the DAC, to prevent overflow of the integer-variable output buffers.

Send the Demodulated or Modulated Signal to the Output Buffer

The final step is to format the processed signal for output to the DAC. When receiving, the *RealOut*() signal is copied, sample by sample, into both the left and right channels. For transmiting, *RealOut*() is copied to the right channel and *ImagOut*() is copied to the left channel of the DAC. If binaural receiving is desired, the *I* and *Q* signal can optionally be sent to the right and left channels respectively, just as in the transmit mode.

Controlling the Demonstration Code

The SDR demonstration code (see Note 3) has a few selected buttons for setting AGC hang time, filter selection and sideband selection. The code for these functions is shown in Fig 18. The code is self-explanatory and easy to modify for additional filters, different hang times and other modes of operation. Feel free to experiment.

The Fully Functional SDR-1000 Software

The SDR-1000, my nomenclature for the PC SDR, contains a significant amount of code not illustrated here. I have chosen to focus this article on the essential DSP code necessary for modulation and demodulation in the frequency domain. As time permits, I hope to write future articles that delve into other interesting aspects of the software design.

Fig 19 shows the completed front-panel display of the SDR-1000. I have had a great deal of fun creating—and modifying many times—this user interface. Most features of the user interface are intuitive. Here are some interesting capabilities of the SDR-1000:

- A real-time spectrum display with one-click frequency tuning using a mouse.
- Dual, independent VFOs with database readout of band-plan allocation. The user can easily access and modify the band-plan database.
- Mouse-wheel tuning with the ability to change the tuning rate with a click of the wheel.
- A multifunction digital- and analog-readout meter for instantaneous and average signal strength, AGC gain, ADC input signal and DAC output signal levels.
- Extensive VFO, band and mode control. The band-switch buttons also provide a multilevel memory on the same band. This means that by pressing a given band button

```
Private Sub cmdAGC_Click(Index As Integer)

    MaxGain = 1000                      'Maximum digital gain = 60dB

    Select Case Index

        Case 0
            AGC = True
            AGCHang = 3                 '3 x 0.04644 sec = 139 ms
        Case 1
            AGC = True
            AGCHang = 7                 '7 x 0.04644 sec = 325 ms
        Case 2
            AGC = False                 'AGC Off
    End Select

End Sub

Private Sub cmdFilter_Click(Index As Integer)

    Select Case Index

        Case 0
            CalcFilter 300, 3000        '2.7KHz Filter
        Case 1
            CalcFilter 500, 1000        '500Hz Filter
        Case 2
            CalcFilter 700, 800         '100Hz Filter
    End Select

End Sub

Private Sub cmdMode_Click(Index As Integer)

    Select Case Index

        Case 0                          'Change mode to USB
            SSB = True
            USB = True
        Case 1                          'Change mode to LSB
            SSB = True
            USB = False
    End Select

End Sub
```

Fig 18 – Control code for the demonstration front panel.

multiple times, it will cycle through the last three frequencies visited on that band.

- Virtually unlimited memory capability is provided through a Microsoft *Access* database interface. The memory includes all key settings of the radio by frequency. Frequencies may also be grouped for scanning.
- Ten standard filter settings are provided on the front panel, plus independent, continuously variable filters for both CW and SSB.
- Local and UTC real-time clock displays.
- Given the capabilities of *Visual Basic*, the possibility for enhancement of the user interface is almost limitless. The hard part is "shooting the engineer" to get him to stop designing and get on the air.

There is much more that can be accomplished in the DSP code to customize the PC SDR for a given application. For example, Leif Åsbrink, SM5BSZ, is doing interesting weak-signal moonbounce work under *Linux*.[16]

Also, Bob Larkin, W7PUA, is using the DSP-10 he first described in the September, October and November 1999 issues of *QST* to experiment with weak-signal, over-the-horizon microwave propagation.[17]

Coming in the Final Article

In the final article, I plan to describe ongoing development of the SDR-1000 hardware. (Note: I plan to delay the final article so that I am able to complete the PC board layout and test the hardware design.) Included will be a tradeoff analysis of gain distribution, noise figure and dynamic range. I will also discuss various approaches to analog AGC and explore frequency control using the AD9854 quadrature DDS.

Several readers have indicated interest in PC boards. To date, all prototype work has been done using "perfboards." At least one reader has produced a circuit board, that person is willing to make boards available to other readers. If you e-mail me, I will gladly put you in contact with those who have built boards. I also plan to have a Web site up and running soon to provide ongoing updates on the project.

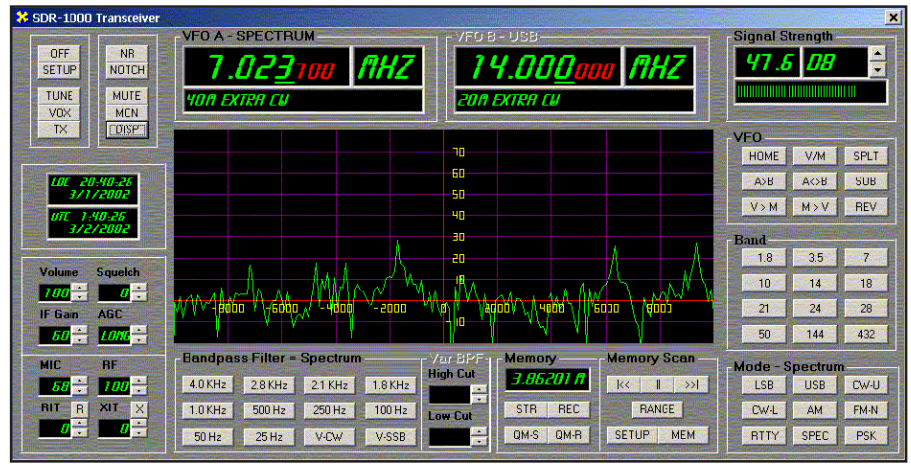


Fig 19—SDR-1000 front-panel display.

Notes

[1]G. Youngblood, AC5OG, "A Software Defined Radio for the Masses, Part 1," *QEX*, Jul/Aug 2002, pp 13-21.

[2]G. Youngblood, AC5OG, "A Software Defined Radio for the Masses, Part 2," *QEX*, Sep/Oct 2002, pp 10-18.

[3]The demonstration source code for this project may be downloaded from *ARRLWeb* at **www.arrl.org/qexfiles/**. Look for 1102Youngblood.zip.

[4]The functions of the Intel Signal Processing Library are now provided in the Intel Performance Primitives (Version 3.0, beta) package for Pentium processors and Itanium architectures. An evaluation copy of IPP is available free to be downloaded from **developer.intel.com/software/products/ipp/ipp30/index. htm**.Commercial use of IPP requires a full license. Do not use IPP with the demo code because it has only been tested on the previous signal processing library.

[5]D. Hershberger, W9GR, and Dr S. Reyer, WA9VNJ, "Using The LMS Algorithm For QRM and QRN Reduction," *QEX*, Sep 1992, pp 3-8.

[6]D. Hall, KF4KL, "Spectral Subtraction for Eliminating Noise from Speech," *QEX*, Apr 1996, pp 17-19.

[7]J. Bloom, KE3Z, "Correlation of Sampled Signals," *QEX*, Feb 1996, pp 24-28.

[8]R. Lyons, *Understanding Digital Signal Processing* (Reading, Massachusetts: Addison-Wesley, 1997) pp 133, 330-340, 429-430.

[9]D. Smith, KF6DX, *Digital Signal Processing Technology* (Newington, Connecticut: ARRL, 2001; ISBN: 0-87259-819-5; Order #8195) pp 4-1 through 4-15.

[10]D. Smith, KF6DX, "Signals, Samples and Stuff: A DSP Tutorial (Part 1)," *QEX* (Mar/Apr 1998), pp 5-6.

[11]Information on FFT convolution may be found in the following references: R. Lyons, *Understanding Digital Signal Processing*, (Addison-Wesley, 1997) pp 435-436; M. Frerking, *Digital Signal Processing in Communication Systems* (Boston, Massachusetts: Kluwer Academic Publishers) pp 202-209; and S. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing* (San Diego, California: California Technical Publishing) pp 311-318.

[12]S. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing* (California Technical Publishing) pp 107-122. This is available for free download at **www.DSPGuide.com**.

[13]Overlap/add method: Ibid, Chapter 18, pp 311-318; M. Freirking, pp 202-209.

[14]S. Smith, Chapter 9, pp 174-177.

[15]M. Frerking, *Digital Signal Processing in Communication Systems*, (Kluwer Academic Publishers) pp 237, 292-297, 328, 339-342, 348.

[16]See Leif Åsbrink's, SM5BSZ, Web site at **ham.te.hik.se/homepage/sm5bsz/**.

[17]See Bob Larkin's, W7PUA, homepage at **www.proaxis.com/~boblark/dsp10.htm**.

A Software Defined Radio for the Masses, Part 4

We conclude this series with a description of a dc-60 MHz transceiver that will allow open-software experimentation with software defined radios.

By Gerald Youngblood, AC5OG

It has been a pleasure to receive feedback from so many *QEX* readers that they have been inspired to experiment with software-defined radios (SDRs) through this article series. SDRs truly offer opportunities to reinvigorate experimentation in the service and attract new blood from the ranks of future generations of computer-literate young people.[1] It is encouraging to learn that many readers see the opportunity to return to a love of experimentation left behind because of the complexity of modern hardware. With SDRs, the opportunity again exists for the experimenter to achieve results that exceed the performance of existing commercial equipment.

Most respondents indicated an interest in gaining access to a complete SDR hardware solution on which they can experiment in software. Based on this feedback, I have decided to offer the SDR-1000 transceiver described in this article as a semi-assembled, three-board set. The SDR-1000 software will also be made available in open-source form along with support for the GNU Radio project on *Linux*.[2] Table 1 outlines preliminary specifications for the SDR-1000 transceiver. I expect to have the hardware available by the time this article is in print.

The ARRL SDR Working Group includes in its mission the encouragement of SDR experimentation through educational articles and the availability of SDR hardware on which to experiment. A significant advance toward this end has been seen in the pages of *QEX* over the last year, and it continues into 2003.

This series began in Part 1 with a general description of digital signal processing (DSP) in SDRs.[3] Part 2 described *Visual Basic* source code to implement a full-duplex, quadrature interface on a PC sound card.[4] Part 3 described the use of DSP to make the PC sound-card interface into a functional software-defined radio.[5] It also explored the filtering technique called FFT fast-convolution filtering. In this final article, I will describe the SDR-1000 transceiver hardware including an analysis of gain distribution, noise figure and dynamic range. There is also a discussion of frequency control using the AD9854 quadrature DDS.

[1]Notes appear on page 28.

8900 Marybank Dr.
Austin, TX 78750
ac5og@arrl.net

To further support the interest generated by this series, I have established a Web site at **home.earthlink.net/~g_youngblood**. As you experiment in this interesting technology, please e-mail suggested enhancements to the site.

Is the "Tayloe Detector" Really New?

In Part 1, I described what I knew at the time about a potentially new approach to detection that was dubbed the "Tayloe Detector." In the same issue, Rod Green described the use of the same circuit in a multiple conversion scheme he called the "Dirodyne".[6] The question has been raised: Is this new technology or rediscovery of prior art? After significant research, I have concluded that both the "Tayloe Detector" and the "Dirodyne" are simply rediscovery of prior art; albeit little known or understood. In the September 1990 issue of *QEX*, D. H. van Graas, PAØDEN, describes "The Fourth Method: Generating and Detecting SSB Signals."[7] The three previous methods are commonly called the *phasing method*, the *filter method* and the *Weaver method*. The "Tayloe Detector" uses *exactly* the same concept as that described by van Grass with the exception that van Grass uses a double-balanced version of the circuit that is actually superior to the singly-balanced detector described by Dan Tayloe[8] in 2001.

In his article, van Graas describes how he was inspired by old frequency-converter systems that used ac motor-generators called "selsyn" motors. The selsyn was one part of an electric axle formerly used in radar systems. His circuit used the CMOS 4052 dual 1-4 multiplexer (an early version of the more modern 3253 multiplexers referenced in Part 1 of this series) to provide the four-phase switching. The article describes circuits for both transmit and receive operation.

Phil Rice, VK3BKR, published a nearly identical version of the van Graas transmitter circuit in *Amateur Radio* (Australia) in February 1998, which may be found on the Web.[9] While he only describes the transmit circuitry, he also states, "... the switching modulator should be capable of acting as a demodulator."

It's the Capacitor, Stupid!

So why is all this so interesting? First, it appears that this truly is a "fourth method" that dates back to at least 1990. In the early 1990s, there was a saying in the political realm: "It's the economy, stupid!" Well, in this case, it's the capacitor, stupid! Traditional commutating mixers do not have capacitors (or integrators) on their output. The capacitor converts the commutating switch from a mixer into a sampling detector (more accurately a track-and-hold) as discussed on page 8 of Part 1 (see Note 3). Because the detector operates according to sampling theory, the mixing products sum aliases back to the same frequency as the difference product, thereby limiting conversion loss. In reality, a switching detector is simply a modified version of a digital commutating filter as described in previous *QEX* articles.[10, 11, 12]

Instead of summing the four or more phases of the commutating filter into a single output, the sampling detector sums the 0° and 180° phases into the in-phase (I) channel and the 90° and 270° phases into the quadrature (Q) channel. In fact, the mathematical analysis described in Mike Kossor's article (see Note 10) applies equally well to the sampling detector.

Is the "Dirodyne" Really New?

The Dirodyne is in reality the sampling detector driving the sampling generator as described by van Graas, forming the architecture first described by Weaver in 1956.[13] The Weaver method was covered in a series of *QEX* articles[14, 15, 16] that are worth reading. Other interesting reading on the subject may be found on the Web in a Phillips Semiconductors application note[17] and an article in *Microwaves & RF*.[18]

Peter Anderson in his Jul/Aug 1999 letter to the *QEX* editor specifically describes the use of back-to-back commutating filters to perform frequency shifting for SSB generation or reception.[19] He states that if, on the output of a commutating filter, we can, "...add a second commutator connected to the same set of capacitors, and take the output from the second commutator. Run the two commutators at different frequencies and find that the input passband is centered at a frequency set by the input commutator; the output passband is centered at a frequency set by the output commutator. Thus, we have a device that shifts the signal frequency, an SSB generator or receiver." This is exactly what the Dirodyne does. He goes on to state, "The frequency-shifting commutating filter is a generalization of the Weaver method of SSB generation."

So What Shall We Call It?

Although Dan Tayloe popularized the sampling detector, it is probably not appropriate to call it the Tayloe detector, since its origin was at least 10 years earlier, with van Graas. Should we call it the "van Graas Detector" or just the "Fourth Method?" Maybe we should, but since I don't know if van Graas originally invented it, I will simply call it the quadrature-sampling detector (QSD) or quadrature-sampling exciter (QSE).

Dynamic Range—How Much is Enough?

The QSD is capable of exceptional dynamic range. It is possible to design a QSD with virtually no loss and 1-dB compression of at least 18 dBm (5 V_{P-P}). I have seen postings on e-mail

Table 1—SDR-1000 Preliminary Hardware Specifications

Frequency Range	0-60 MHz
Minimum Tuning Step	1 μHz
DDS Clock	200 MHz, <1 ps RMS jitter
1dB Compression	+6 dBm
Max. Receive Bandwidth	44 kHz-192 kHz (depends on PC sound card)
Transmit Power	1 W PEP
PC Control Interface	PC parallel port (DB-25 connector)
Rear Panel Control Outputs	7 open-collector Darlington outputs
Input Controls	PTT, Code Key, 2 Spare TTL Inputs
Sound Card Interface	Line in, Line out, Microphone in
Power	13.8 V dc

Table 2—Acceptable Noise Figure for Terrestrial Communications

Frequency (MHz)	*Acceptable NF (dB)*
1.8	45
3.5	37
4.0	27
14.0	24
21.0	20
28.0	15
50.0	9
144.0	2

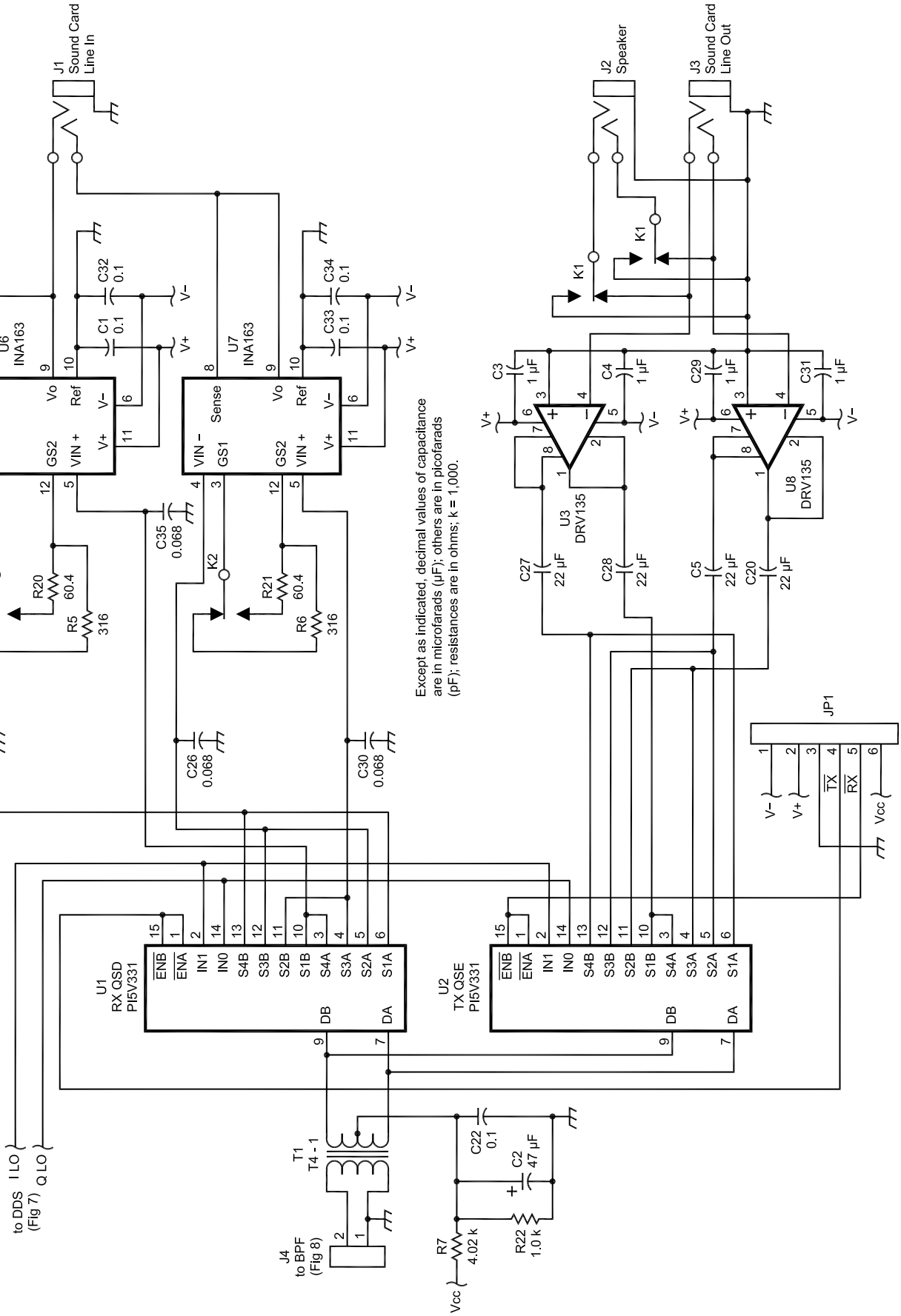


Fig 1—SDR-1000 receiver/exciter schematic.

reflectors claiming measured *IP3* in the +40 dBm range for QSD detectors using 5-V parts. With ultra-low-noise audio op amps, it is possible to achieve an analog noise figure on the order of 1 dB without an RF preamplifier. With appropriately designed analog AGC and careful gain distribution, it is theoretically possible to achieve over 150 dB of total dynamic range. The question is whether that much range is needed for typical HF applications. In reality, the answer is no. So how much is enough?

Several *QEX* writers have done an excellent job of addressing the subject.[20, 21, 22] Table 2 was originally published in an October 1975 *ham radio* article.[23] It provides a straightforward summary of the acceptable receiver noise figure for terrestrial communication for each band from 160 m to 2 m. Table 3 from the same article illustrates the acceptable noise figures for satellite communications on bands from 10 m to 70 cm.

For my objective of dc-60 MHz coverage in the SDR-1000, Table 2 indicates that the acceptable noise figure ranges from 45 dB on 160 m to 9 dB on 6 m. This means that a 1-dB noise figure is overkill until we operate near the 2-m band. Further, to utilize a 1-dB noise figure requires almost 70 dB of analog gain ahead of the sound card. This means that proper gain distribution and analog AGC design is critical to maximize IMD dynamic range.

After reading the referenced articles and performing measurements on the Turtle Beach Santa Cruz sound card, I determined that the complexity of an analog AGC circuit was unwarranted for my application. The Santa Cruz card has an input clipping level of 12 V (RMS, 34.6 dBm, normalized to 50 Ω) when set to a gain of –10 dB. The maximum output available from my audio signal generator is 12 V (RMS). The SDR software can easily monitor the peak signal input and set the corresponding sound card input gain to effectively create a digitally controlled analog AGC with no external hardware. I measured the sound card's 11-kHz SNR to be in the range of 96 dB to 103 dB, depending on the setting of the card's input gain control. The input control is capable of attenuating the gain by up to 60 dB from full scale. Given the large signal-handling capability of the QSD and sound card, the 1-dB compression point will be determined by the output saturation level of the instrumentation amplifier.

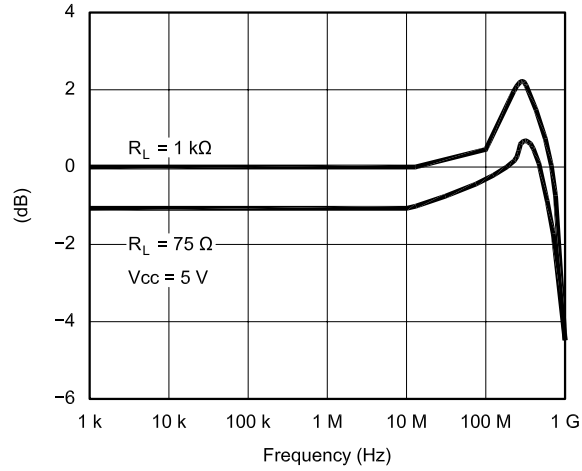
Of note is the fact that DVD sales are driving improvements in PC sound cards. The newest 24-bit sound cards sample at a rate of up to 192 kHz. The Waveterminal 192X from EGO SYS is one example.[24] The manufacturer boasts of a 123 dB dynamic range, but that number should be viewed with caution because of the technical difficulties of achieving that many bits of true resolution. With a 192-kHz sampling rate, it is possible to achieve real-time reception of 192 kHz of spectrum (assuming quadrature sampling).

Quadrature Sampling Detector/ Exciter Design

In Part 1 of this series (Note 3), I described the operation of a single-balanced version of the QSD. When the circuit is reversed so that a quadrature excitation signal drives the sampler, a SSB generator or exciter is created. It is a simple matter to reverse the SDR receiver software so that it transforms microphone input into filtered, quadrature output to the exciter.

While the singly-balanced circuit described in Part 1 is extremely simple, I have chosen to use the double-balanced QSD as shown in Fig 1 because of its superior common mode and even-harmonic rejection. U1, U6 and U7 form the receiver and U2, U3 and U8 form the exciter. In the receive mode, the QSD functions as a two-capacitor commutating filter, as described by Chen Ping in his article (Note 11). A commutating filter works like a comb filter, wherein the circuit responds to harmonics of the commutation frequency. As he notes, ". . . it can be shown that signals having harmonic numbers equal to any of the integer factors of the number of capacitors may pass." Since two capacitors are used in each of the I and Q channels, a two-capacitor commutating filter is formed. As Ping further states, this serves to suppress the even-order harmonic responses of the circuit. The output of a two-capacitor filter is extremely phase-sensitive, therefore allowing the circuit to perform signal detection just as a CW demodulator does. When a signal is near the filter's center frequency, the output amplitude would be modulated at the difference (beat) frequency. Unlike a typical filter, where phase sensitivity is undesirable, here we actually take advantage of that capability.

The commutator, as described in Part 1, revolves at the center frequency of the filter/detector. A signal tuned exactly to the commutating frequency will result in a zero beat. As the signal is tuned to either side of the commutation frequency, the beat note output will be proportional to the difference frequency. As the signal is tuned toward the second harmonic, the output will decrease until a null occurs at the harmonic frequency. As the signal is tuned



Note: Insertion Log = 20 log | Vo / Vs |

Fig 2—QS4A210 insertion loss versus frequency

Table 3—Acceptable Noise Figure for Satellite Communications

Frequency (MHz)	*Galactic Noise Floor (dBm/Hz)*	*Acceptable NF (dB)*
28	–125	8
50	–130	5
144	–139	1
220	–140	0.7
432	–141	0.2

further, it will rise to a peak at the third harmonic and then decrease to another null at the fourth harmonic. This cycle will repeat indefinitely with an amplitude output corresponding to the $\sin(x)/x$ curve that is characteristic of sampling systems as discussed in DSP texts. The output will be further attenuated by the frequency-response characteristics of the device used for the commutating switch. The PI5V331 multiplexer has a 3-dB bandwidth of 150 MHz. Other parts are available with 3-dB bandwidths of up to 1.4 GHz (from IDT Semiconductor).

Fig 2 shows the insertion loss versus frequency for the QS4A210. The upper frequency limitation is determined by the switching speed of the part (1 ns = T_{on} / T_{off}, best-case or 12.5 ns worst-case for the 1.4-GHz part) and the $\sin(x)/x$ curve for under-sampling applications.

The PI5V331 (functionally equivalent to the IDT QS4A210) is rated for analog operation from 0 to 2 V. The QS4A210 data sheet provides a drain-to-source on-resistance curve versus the input voltage as shown in Fig 3. From the curve, notice that the on resistance (R_{on}) is linear from 0 to 1 V and increases by less than 2 Ω at 2 V. No curve is provided in the PI5V331 data sheet, but we should be able to assume the two are comparable. In fact, the PI5V331 has a R_{on} specification of 3 Ω (typical) versus the 5 Ω (typical) for the QS41210. In the receive application of the QSD, the R_{on} is looking into the 60-MΩ input of the instrumentation amplifier. This means that ΔR_{on} modulation is virtually nonexistent and will have no material effect on circuit linearity.[25] Unlike typical mixers, which are nonlinear, the QSD is a linear detector!

Eq 1 determines the bandwidth of the QSD, where R_{ant} is the antenna impedance, C_S is the sampling capacitor value and n is the total number of sampling capacitors ($1/n$ is effectively the switch duty cycle on each capacitor). In the doubly balanced QSD, n is equal to 2 instead of 4 as in the singly balanced circuit. This is because the capacitor is selected twice during each commutation cycle in the doubly balanced version.

$$BW_{det} = \frac{1}{\pi n R_{ant} C_S} \qquad \text{(Eq 1)}$$

A tradeoff exists in the choice of QSD bandwidth. A narrow bandwidth such as 6 kHz provides increased blocking and IMD dynamic range because of the very high Q of the circuit. When designed for a 6-kHz bandwidth, the response at 30 kHz—one decade from the 3-kHz 3-dB point—either side of the center frequency will be attenuated by 20 dB. In this case, the QSD forms a 6-kHz-wide tracking filter centered at the commutating frequency. This means that strong signals outside the passband of the QSD will be attenuated, thereby dramatically increasing *IP3* and blocking dynamic range.

I am interested in wider bandwidth for several reasons and therefore willing to trade off some of the IMD-reduction potential of the QSD filter. In SDR applications, it is desirable in many cases to receive the widest bandwidth of which the sound card is capable. In my original design, that is 44 kHz with quadrature sampling. This capability increases to 192 kHz with the newest sound cards. Not only does this allow the capability of observing the real-time spectrum of up to 192 kHz, but it also brings the potential for sophisticated noise and interference reduction.[26]

Further, as we will see in a moment, the wider bandwidth allows us to reduce the analog gain for a given sensitivity level. The 0.068-µF sampling capacitors are selected to provide a QSD bandwidth of 22 kHz with a 50-Ω antenna. Notice that any variance in the antenna impedance will result in a corresponding change in the bandwidth of the detector. The only way to avoid this is to put a buffer in front of the detector.

The receiver circuit shown in Part 1 used a differential summing op amp after the detector. The primary advantage of a low-noise op amp is that it can provide a lower noise figure at low gain settings. Its disadvantage is that the inverting input of the op amp will be at virtual ground and the non-inverting input will be high impedance. This means that the sampling capacitor on the inverting input will be loaded differently from the non-inverting input. Thus, the respective passbands of the two inputs will not track one another. This problem is eliminated if an instrumentation amplifier is used. Another advantage of using an instrumentation amplifier as opposed to an op amp is that the antenna impedance is removed from the amplifier gain equation. The single disadvantage of the instrumentation amplifier is that the voltage noise

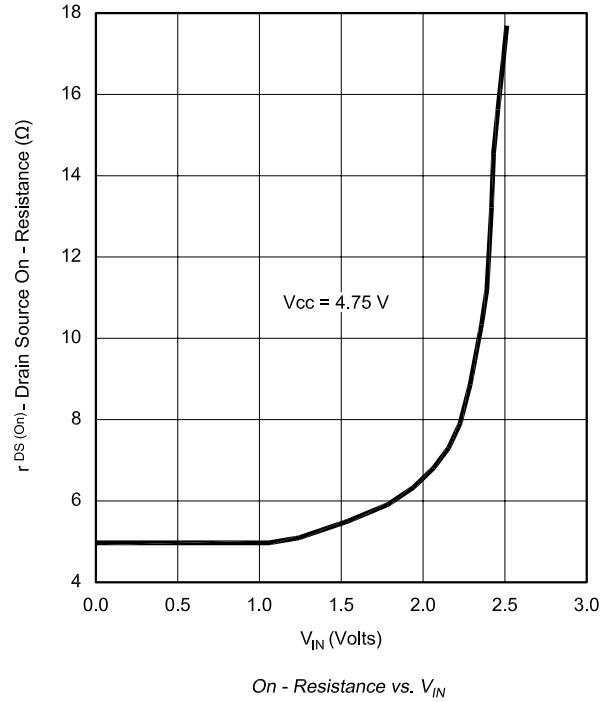


Fig 3—QS4A210 R_{on} versus V_{IN}.

Table 4—INA 163 Noise Data at 10 kHz

Gain (dB)	*e_n*	*i_n*	*NF (dB)*
20	7.5 nV/√ Hz	0.8 pA/√ Hz	12.4
40	1.8 nV/√ Hz	0.8 pA/√ Hz	3.0
60	1.0 nV/√ Hz	0.8 pA/√ Hz	1.3

and thus the noise figure increases with decreasing gain.

Table 4 shows the voltage noise, current noise and noise figure for a 200-Ω source impedance for the TI INA163 instrumentation amplifier. Since a single resistor sets the gain of each amplifier, it is a simple matter to provide two or more gain settings with relay or solid-state switching.

Unlike typical mixers, which are normally terminated in their characteristic impedances, the QSD is a high-impedance, sampling device. Within the passband, the QSD outputs are terminated in the 60-MΩ inputs of the instrumentation amplifiers. The IDT data sheet for the QS4A210 indicates that the switch has *no insertion loss* with loads of 1 kΩ or more! This coincides with my measurements on the circuit. If you apply 1 V of RF into the detector, you get 1 V of audio out on each of the four capacitors—a no-loss detector. Outside the passband, the decreasing reactance of the sampling capacitors will reduce the signal level on the amplifier inputs. While it is possible to insert series resistors on the output of the QSD, so that it is terminated outside the passband, I believe this is unnecessary. For receive operation, filter reflections outside the passband are not very important. Further, the termination resistors would create an additional source of thermal noise.

As stated earlier, the circuitry of the QSD may be reversed to form a quadrature sampling exciter (QSE). To do so, we must differentially drive the I and Q inputs of the QSE. The Texas Instruments DRV135 50-Ω differential audio line driver is ideally suited for the task. Blocking capacitors on the driver outputs prevent dc-offset variation between the phases from creating a carrier on the QSE output. Carrier suppression has been measured to be on the order of –48 dBc relative to the exciter's maximum output of +10 dBm. In transmit mode, the output impedance of the exciter is 50 Ω so that the band-pass filters are properly terminated.

Conveniently, T/R switching is a simple matter since the QSD and QSE can have their inputs connected in parallel to share the same transformer. Logic control of the respective multiplexer-enable lines allows switching between transmit and receive mode.

Level Analysis

The next step in the design process is to perform a system-level analysis of the gain required to drive the sound card A/D converter. One of the better references I have found on the subject is the book by W. Sabin and E. Schoenike, *HF Radio Systems and Circuits*.[27] The book includes an *Excel* spreadsheet that allows interactive examination of receiver performance using various A/D converters, sample rates, bandwidths and gain distributions. I have placed a copy of the SDR-1000 Level Analysis spreadsheet (by permission, a highly modified version of the one provided in the book) for download from ARRLWeb.[28] Another excellent resource on the subject is the Digital Receiver/Exciter Design chapter from the book *Digital Signal Processing in Communication Systems*.[29]

Notice that the former reference has a better discussion of the minimum gain required for thermal noise to transition the quantizing level as discussed here. Neither text deals with the effects of atmospheric noise on the noise floor and hence on dynamic range. This is—in my opinion—a major oversight for HF communications since atmospheric noise will most likely limit the minimum discernable signal, not thermal noise.

For a weak signal to be recovered, the minimum analog gain must be great enough so that the weakest signal to be received, plus thermal and atmospheric noise, is greater than at least one A/D converter quantizing level (the least-significant usable bit). For the A/D converter quantizing noise to be evenly distributed, several quantizing levels must be traversed. There are two primary ways to achieve this: Out-of-band dither noise may be added and then filtered out in the DSP routines, or in-band thermal and atmospheric noise may be amplified to a level that accomplishes the same. While the first approach offers the best sensitivity at the lowest gain, the second approach is simpler and was chosen for my application. *HF Radio Systems and Circuits* states, "Normally, if the noise is Gaussian distributed, and the RMS level of the noise at the A/D converter is greater than or equal to the level of a sine wave which just bridges a single quantizing level, an adequate number of quantizing levels will be bridged to guarantee uniformly distributed quantizing noise." Assuming uniform noise distribution, Eq 2 is used to determine the quantizing noise density, N_{0q}:

$$N_{0q} = \frac{\left(\frac{V_{pp}}{2^b}\right)^2}{6 f_s R} \text{ W/Hz} \qquad \text{(Eq 2)}$$

where
$V_{P\text{-}P}$= peak-to-peak voltage range
b = number of valid bits of resolution
f_s = A/D converter sampling rate
R = input resistance
N_{0q} = quantizing noise density

The quantizing noise decreases by 3 dB when doubling the sampling rate and by 6 dB for every additional bit of resolution added to the A/D converter. Notice that just because a converter is specified to have a certain number of bits does not mean that they are all usable bits. For example, a converter may be specified to have 16 bits; but in reality, only be usable to 14-bits. The Santa Cruz card utilizes an 18-bit A/D converter to deliver 16 usable bits of resolution. The maximum signal-to-noise ratio may be determined from Eq 3:

$$SNR = 6.02b + 1.75 \text{ dB} \qquad \text{(Eq 3)}$$

For a 16-bit A/D converter having a maximum signal level (without input attenuation) of 12.8 $V_{P\text{-}P}$, the minimum quantum level is –70.2 dBm. Once the quantizing level is known, we can compute the minimum gain required from Eq 4:

$$Gain = quantizing\ level - kTB + analog\ NF + atmospheric\ NF - 10\log_{10} BW \qquad \text{(Eq 4)}$$

where:

$$quantizing\ level = 10\log_{10}\left(\frac{\left(\frac{V_{pp}\times 0.707}{2^b\times 2}\right)^2}{50\times 0.001}\right) \text{dBm}$$

kTB / Hz = –174 dBm/Hz
analog NF = analog receiver noise figure, in decibels
atmospheric NF = atmospheric noise figure for a given frequency
BW = the final receive filter bandwidth in hertz

Table 5, from the SDR-1000 Level Analysis spreadsheet, provides the cascaded noise figure and gain for the circuit shown if Fig 1. This is where things get interesting.

Fig 4 shows an equivalent circuit for the QSD and instrumentation amplifier during a respective switch period. The transformer was selected to have a 1:4 impedance ratio. This means that the turns ratio from the primary to the secondary for each switch to ground is 1:1, and therefore the voltage on each switch is equal to the input signal voltage. The differential impedance across the transformer secondary will be 200 Ω, providing a good noise match to the INA163 amplifier. Since the input impedance of the INA163 is 60 MΩ, power loss through the circuit is virtually nonexistent. We must therefore analyze the circuit based on voltage gain, not power gain.

Table 5 –Cascaded Noise Figure and Gain Analysis from the SDR-1000 Level Analysis Spreadsheet

		BPF	*T1-4*	*PI5V331*	*INA163*	*ADC*
dB	Noise Figure	0.0	0.0	0.0	3.0	58.6
dB	Gain	0.0	6.0	0.0	40.0	0.0
Equivalent Power Factor	Noise Factor	1.00	1.00	1.00	1.99	720,482
Equivalent Power Factor	Gain	1	4	1	10,000	1
Clipping Level	Vpk			1.0	13.0	6.4
Clipping Level	dBm			10.0	32.3	26.1
Cascaded Gain	dB	0.0	6.0	6.0	46.0	46.0
Cascaded Noise Factor		1.00	1.00	1.00	1.25	19.06
Cascaded Noise Figure	dB	0.0	0.0	0.0	1.0	12.8
Output Noise	dBm/Hz	–174.0	–174.0	–174.0	–173.0	–161.2

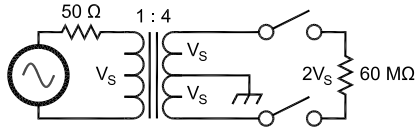


Fig 4—Doubly balanced QSD equivalent circuit.

Table 6—Atmospheric Equivalent Noise Figure By Band

Band (Meters)	*Ext Noise (dBm/Hz)*	*Ext NF (dB)*
160	–128	46
80	–136	38
40	–144	30
30	–146	28
20	–146	28
17	–152	22
15	–152	22
12	–154	20
10	–156	18
6	–162	12

Table 7—SDR-1000 Level Analysis Assumptions for the 10-Meter Band with 40 dB of INA Gain

Receiver Gain Distribution and Noise Performance	
Turtle Beach Santa Cruz Audio Card	
Band Number	9
Band	10 Meters
Include External NF? (True=1, False=0)	1
External (Atmospheric) Noise Figure	18 dB
A/D Converter Resolution (bits)	16 bits (98.1 dB)
A/D Converter Full–Scale Voltage	6.4 V-peak (26.1 dBm)
A/D Converter Quantizing Signal Level	–70.2 dBm
Quantizing Gain Over/(Under)	7.2 dB
A/D Converter Sample Frequency	44.1 kHz
A/D Converter Input Bandwidth (BW1)	40.0 kHz
Information Bandwidth (BW2)	0.5 kHz
Signal at Antenna for INA Saturation	–13.7 dBm
Nominal DAC Output Level	0.5 V peak (4.0 dBm)
AGC Threshold at Ant (40 dB Headroom)	–51.4 dBm
Sound Card AGC Range	60.0 dB

That means that we get a 6-dB differential voltage gain from the input transformer—the equivalent of a 0-dB noise figure amplifier! Further, there is no loss through the QSD switches due to the high-impedance load of the INA. With a source impedance of 200 Ω, the INA163 has a noise figure of approximately 12.4 dB at 20 dB of gain, 3 dB at 40 dB of gain and 1.3 dB at 60 dB of gain.

In fact, the noise figure of the analog front end is so low that if it were not for the atmospheric noise on the HF bands, we would need to add a lot of gain to amplify the thermal noise to the quantizing level. The textbook references ignore this fact. In addition to the *ham radio* article (Note 23) and Peter Chadwick's *QEX* article (Note 20), John Stephenson in his *QEX* article[30] about the ATR-2000 HF transceiver provides further insight into the subject. Table 6 provides a summary of the external noise figure for a by-band quiet location as determined from Fig 1 in Stephenson's article. As can be seen from the table, it is counterproductive to have high gain and low receiver noise figure on most of the HF bands.

Tables 7 and 8 are derived from the *SDR-1000 Level Analysis* spreadsheet (Note 28) for the 10-m band. The spreadsheet tables interact with one another so that a change in an assumption will flow through all the other tables. A detailed discussion of the spreadsheet is beyond the scope of this text. The best way to learn how to use the spreadsheet is to plug in values of your own. It is also instructive to highlight cells of interest to see how the formulas are derived. Based on analysis using the spreadsheet, I have chosen to make the gain setting relay-selectable between INA gain settings of 20 dB for the lower bands and 40 dB for the higher bands.

It is important to remember that my noise and dynamic-range calculations include external noise figure in addition to the thermal noise figure. This is much more realistic for HF applications than the typical lab testing and calculations you see in most references. With the INA163 gain set to 40 dB, the cascaded analog thermal NF is calculated to be just 1 dB at the input to the sound card. If it were not for the external noise, nearly 70 dB of analog gain would be required to amplify the thermal noise

Table 8—SDR-1000 Level Analysis Detail for the 10-Meter Band with 40dB of INA Gain

Antenna Signal Level (dBm)	*INA Output Level (dBm)*	*Antenna Overload Level (dBm)*	*Total Analog Gain (dB)*	*Sound Card AGC Reduction (dB)*	*Noise at A/D Input in BW1 (dBm)*	*A/D Signal Level (dBm)*	*Noise in A/D Input in BW2 (dBm)*	*Quantizing Noise of A/D in BW2 (dBm)*	*Total Noise in BW2 (dBm)*	*Output S/N Ratio in BW2 (dB)*	*Digital Gain Required (dB)*
–128	–82		46.0	0.0	–63.0	–82.0	–82.0	–88.4	–81.1	–0.9	86.0
–118	–72		46.0	0.0	–63.0	–72.0	–82.0	–88.4	–81.1	9.1	76.0
–108	–62		46.0	0.0	–63.0	–62.0	–82.0	–88.4	–81.1	19.1	66.0
–98	–52		46.0	0.0	–63.0	–52.0	–82.0	–88.4	–81.1	29.1	56.0
–88	–42		46.0	0.0	–63.0	–42.0	–82.0	–88.4	–81.1	39.1	46.0
–78	–32		46.0	0.0	–63.0	–32.0	–82.0	–88.4	–81.1	49.1	36.0
–68	–22		46.0	0.0	–63.0	–22.0	–82.0	–88.4	–81.1	59.1	26.0
–58	–12		46.0	0.0	–63.0	–12.0	–82.0	–88.4	–81.1	69.1	16.0
–48	–2		42.7	–3.3	–66.4	–5.4	–85.4	–88.4	–83.6	78.2	9.4
–38	8		32.7	–13.3	–76.4	–5.4	–95.4	–88.4	–87.6	82.2	9.4
–28	18		22.7	–23.3	–86.4	–5.4	–105.4	–88.4	–88.3	82.9	9.4
–18	28		12.7	–33.3	–96.4	–5.4	–115.4	–88.4	–88.4	83.0	9.4
–8	38	6	2.7	–43.3	–106.4	–5.4	–125.4	–88.4	–88.4	83.0	9.4
2	48	16	–7.3	–53.3	–116.4	–5.4	–135.4	–88.4	–88.4	83.0	9.4
12	58	26	–14.0	–60.0	–123.0	–2.0	–142.0	–88.4	–88.4	86.4	6.0
22	68	36	–14.0	–60.0	–123.0	8.0	–142.0	–88.4	–88.4	96.4	–4.0
32	78	46	–14.0	–60.0	–123.0	18.0	–142.0	–88.4	–88.4	106.4	–14.0

to the quantizing level or dither noise would have to be added outside the passband. Fig 6 illustrates the signal-to-noise ratio curve with external noise for the 10-m band and 40 dB of INA gain. Fig 5 shows the same curve without external noise and with INA gain of 60 dB. This much gain would not improve the sensitivity in the presence of external noise but would reduce blocking and IMD dynamic range by 20 dB. On the lower bands, 20 dB or lower INA gain is perfectly acceptable given the higher external noise.

Frequency Control

Fig 7 illustrates the Analog Devices AD9854 quadrature DDS circuitry for driving the QSD/QSE. Quadrature local-oscillator signals allow the elimination of the divide-by-four Johnson counter, described in Part 1, so that the DDS runs at the carrier frequency instead of its fourth harmonic. I have chosen to use the 200-MHz version of the part to minimize heat dissipation, and because it easily meets my frequency coverage requirements of dc-60 MHz. The DDS outputs are connected to seventh-order elliptical low-pass filters that also provide a dc reference for the high-speed comparators. The AD9854 may be controlled either through a SPI port or a parallel interface. There are timing issues in SPI mode that require special care in programming. Analog Devices have developed a protocol that allows the chip to be put into external I/O update mode to work around the serial

About Intel Performance Primitives

Many readers have inquired about Intel's replacement of its Signal Processing Library (SPL) with the Intel Performance Primatives (IPP). The SPL was a free distribution, but the Intel Web site states that IPP requires payment of a $199 fee after a 30 day evaluation period. A fully functional trial version of IPP may be downloaded from the Intel site at **www.intel.com/software/products/global/eval.htm**. The author has confirmed with Intel Product Management that no license fee is required for amateur experimentation using IPP, and there is no limit on the evaluation period for such use. Intel actually encourages this type of experimental use. Payment of the license fee is required if and only if there is a commercial distribution of the DLL code.—*Gerald Youngblood*

timing problem. In the final circuit, I chose to use the parallel mode.

According to Peter Chadwick's article (Note 20), phase-noise dynamic range is often the limiting factor in receivers instead of IMD dynamic range. The AD9854 has a residual phase noise of better than –140 dBc/Hz at a 10-kHz offset when directly clocked at 300 MHz and programmed for an 80-MHz output. A very low-jitter clock oscillator is required so that the residual phase noise is not degraded significantly.

High-speed data communications technology is fortunately driving the introduction of high-frequency crystal oscillators with very low jitter specifications. For example, Valpey Fisher makes oscillators specified at less than 1 ps RMS jitter that operate in the desired 200-300 MHz range. According to Analog Devices, 1 ps is on the order of the residual jitter of the AD9854.

Band-Pass Filters

Theoretically, the QSD will work just fine with low-pass rather than band-pass filters. It responds to the carrier frequency and odd harmonics of the carrier; however, very large signals at half the carrier frequency can be heard in the output. For example, my measurements show that when the receiver is tuned to 7.0 MHz, a signal at 3.5 MHz is attenuated by 49 dB. The measurements show that the attenuation of the second harmonic is 37 dB and the third harmonic is down 9 dB from the 7-MHz reference. While a simple low-pass filter will suffice in some applications, I chose to use band-pass filters.

Fig 8 shows the six-band filter design for the SDR-1000. Notice that only the 2.5-MHz filter has a low-pass characteristic; the rest are band-pass filters.

SDR-1000 Board Layout

For the final PC-board layout, I decided on a 3×4-inch form factor. The receiver, exciter and DDS are located on one board. The band-pass filter and a 1-W driver amplifier are located on a second board. The third board has a PC parallel-port interface for control, and power regulators for operation from a 13.8-V dc power source. The three boards sandwich together into a small 3×4×2-inch module with rear-mount connectors and no interconnection wiring required. The boards use primarily surface-mount components, except for the band-pass filter, which uses mostly through-hole components.

Acknowledgments

I would like to thank David Brandon and Pascal Nelson of Analog Devices for their answering my questions about the AD9854 DDS. My appreciation also goes to Mike Pendley, WA5VTV, for his assistance in design of the band-pass filters as well as his ongoing advice.

Conclusion

This series has presented a practical approach to high-performance SDR development that is intended to spur broad-scale amateur experimentation. It is my hope—and that of the ARRL SDR Working Group—that many will be encouraged to contribute to the technical art in this fascinating area. By making the SDR-1000 hardware and software available to the amateur community, software extensions may be easily and quickly added. Thanks for reading.

Notes

[1]M. Markus, N3JMM, "*Linux*, Software Radio and the Radio Amateur," *QST*, October 2002, pp 33-35.

[2]The GNU Radio project may be found at **www.gnu.org/software/gnuradio/gnuradio.html**.

[3]G. Youngblood, AC5OG, "A Software Defined Radio for the Masses: Part 1," *QEX*, Jul/Aug 2002, pp 13-21.

[4]G. Youngblood, AC5OG, "A Software Defined Radio for the Masses: Part 2," *QEX*, Sep/Oct 2002, pp 10-18.

[5]G. Youngblood, AC5OG, "A Software Defined Radio for the Masses: Part 3," *QEX*, Nov/Dec 2002, pp 27-36.

[6]R. Green, VK6KRG, "The Dirodyne: A New Radio Architecture?" *QEX*, Jul/Aug 2002, pp 3-12.

[7]D. H. van Graas, "The Fourth Method: Generating and Detecting SSB Signals," *QEX*, Sep 1990, pp 7-11.

[8]D. Tayloe, N7VE, "Letters to the Editor, Notes on 'Ideal' Commutating Mixers (Nov/Dec 1999)," *QEX*, Mar/Apr 2001, p 61.

[9]P. Rice, VK3BHR, "SSB by the Fourth Method?" **ironbark.bendigo.latrobe.edu.au/~rice/ssb/ssb.html**.

[10]M. Kossor, WA2EBY, "A Digital Commutating Filter," *QEX*, May/Jun 1999, pp 3-8.

[11]C. Ping, BA1HAM, "An Improved Switched Capacitor Filter," *QEX*, Sep/Oct 2000, pp 41-45.

[12]P. Anderson, KC1HR, "Letters to the Editor, A Digital Commutating Filter," *QEX*, Jul/Aug 1999, pp 62.

[13]D. Weaver, "A Third Method of Generation of Single-Sideband Signals," *Proceedings of the IRE*, Dec 1956.

[14]P. Anderson, KC1HR, "A Different Weave of SSB Exciter," *QEX*, Aug 1991, pp 3-9.

[15]P. Anderson, KC1HR, "A Different Weave of SSB Receiver," *QEX*, Sep 1993, pp 3-7.

[16]C. Puig, KJ6ST, "A Weaver Method SSB Modulator Using DSP," *QEX*, Sep 1993, pp 8-13.

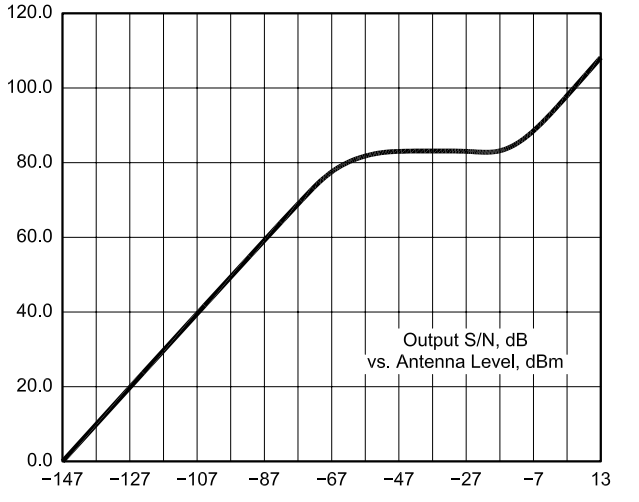


Fig 5—Output signal-to-noise ratio excluding external (atmospheric) noise. INA gain is set to 60 dB. Antenna signal level for saturation is –33.7 dBm.

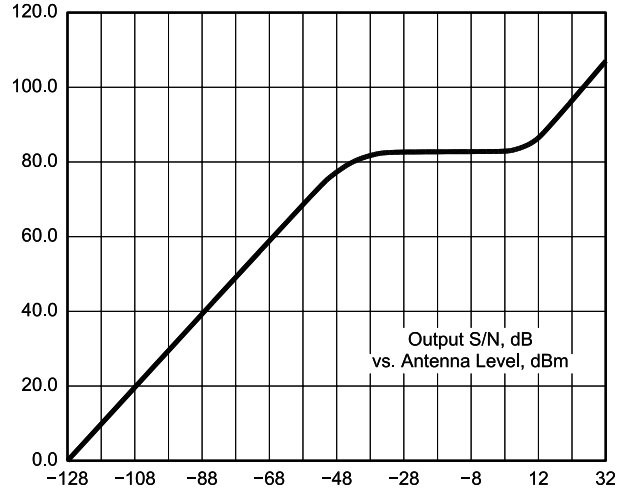


Fig 6—Output signal-to-noise ratio for the 10-m band including external (atmospheric) noise. INA gain is set to 40 dB. Antenna signal level for INA saturation is –13.7dBm.

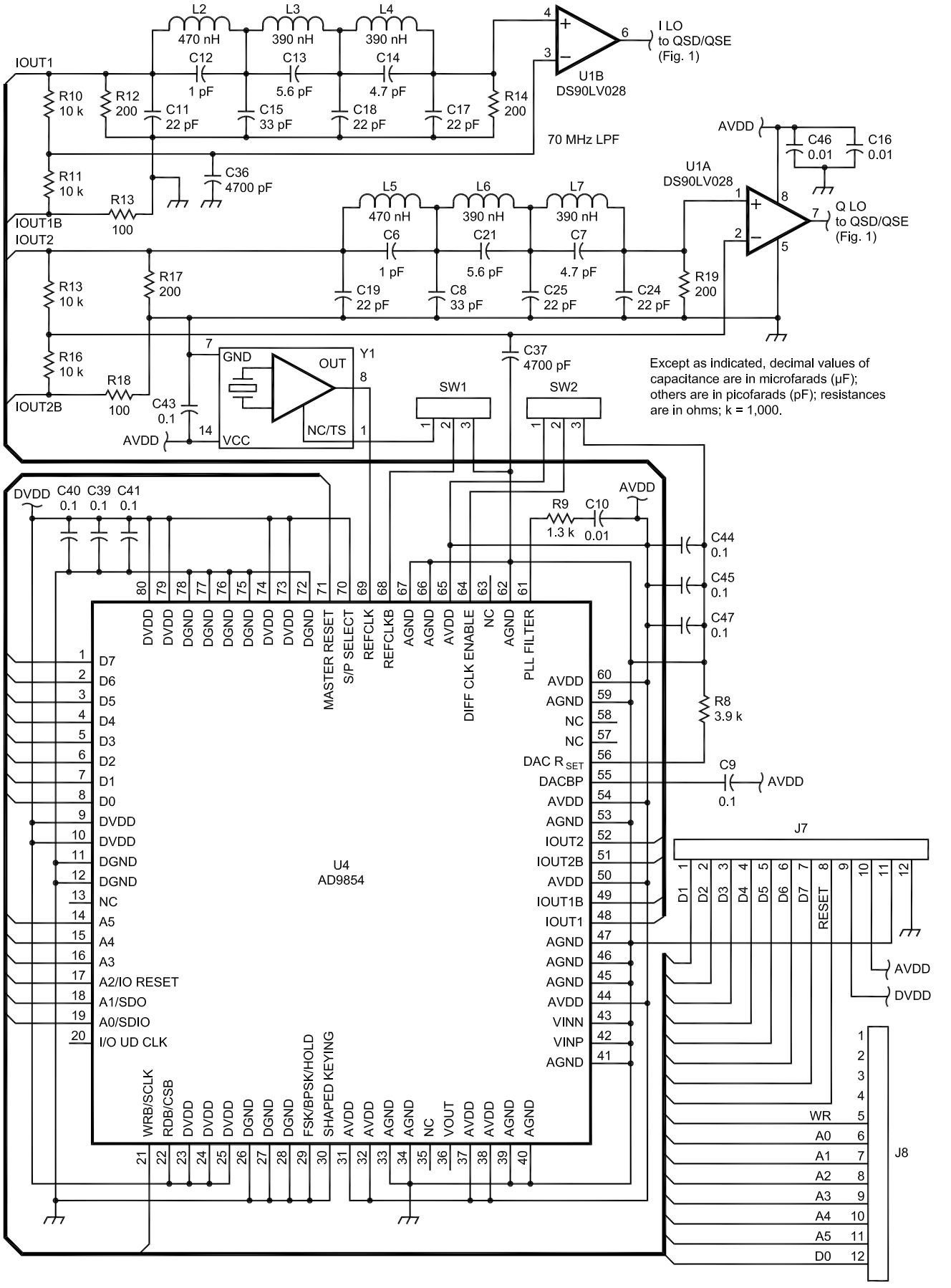


Fig 7—SDR-1000 quadrature DDS schematic.

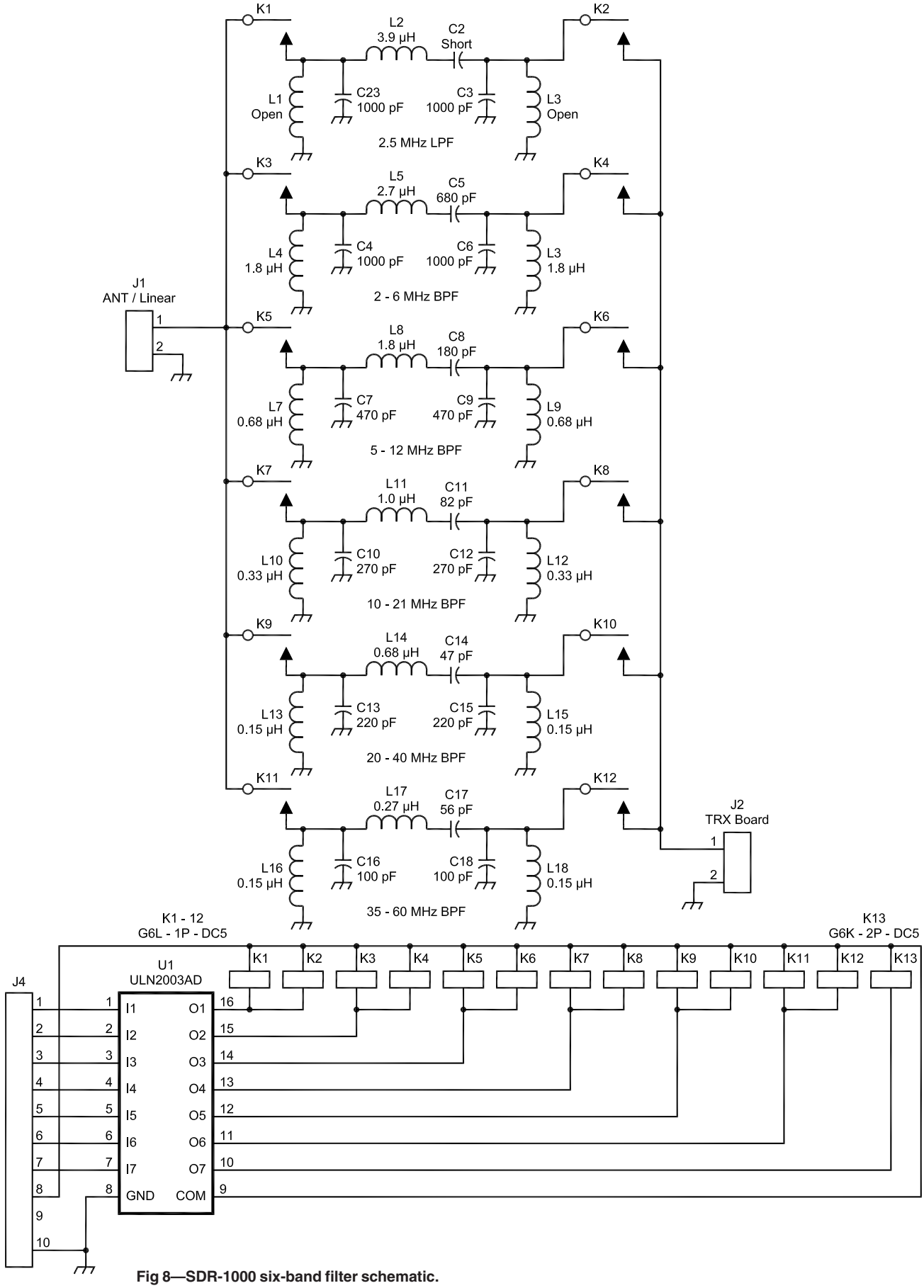


Fig 8—SDR-1000 six-band filter schematic.

[17]R. Zavrell Jr, "New Low Power Single Sideband Circuits," Phillips Semiconductors Application Note AN1981, Oct 1997, **www.semiconductors.philips.com/acrobat/applicationnotes/AN1981.pdf**.
[18]M. Vidmar, "Achieve Phasor Rotation With Analog Switches," *Microwaves & RF*, Feb 2000, **www.mwrf.com/Articles/Index.cfm?ArticleID=10069&extension=pdf**.
[19]P. Anderson, KC1HR, "Letters to the Editor, A Digital Commutating Filter," *QEX*, Jul/Aug 1999, pp 62.
[20]P. Chadwick, G3RZP, "HF Receiver Dynamic Range: How Much Do we Need?" *QEX,* May/Jun 2002, pp 36-41.
[21]J. Scarlett, KD7O, "A High-Performance Digital Transceiver Design, Part 1," *QEX*, Jul/Aug 2002, pp 36-39.
[22]U. Rohde, KA2WEU/DJ2LR/HB9AWE, "Theory of Intermodulation and Reciprocal Mixing: Practice, Definitions and Measurements in Devices and Systems, Part 1," *QEX*, Nov/Dec 2002, pp 3-15.
[23]J. Fisk, "Receiver Sensitivity, Noise Figure and Dynamic Range," *ham radio* magazine, Oct 1975, pp 8-25. *Ham radio* magazine is available on CD from ARRL.
[24]The Waveterminal 192X is manufactured by EGO SYS and may be found on the Web at **www.esi-pro.com**.
[25]J. Wynne, "R_{on} Modulation in CMOS Switches and Multiplexers; What It is and How to Predict its Effect on Signal Distortion," Application Note AN-251, Analog Devices, pp 1-3; **www.analog.com/UploadedFiles/Application_Notes/413221855AN251.pdf**.
[26]L. Åsbrink, SM5BSZ, "*Linrad*: New Possibilities for the Communications Experimenter, Part 1," *QEX,* Nov/Dec 2002, pp 37-41. More information on the Web at **www.nitehawk.com/sm5bsz/linuxdsp/linrad.htm**.
[27]W. Sabin and E. Schoenike, Editors, *HF Radio Systems & Circuits*, (Atlanta, Georgia: Noble Publishing Corporation, ISBN 1-884932-04-5) pp 349-355 and 642-644.
[28] You can download this package from the ARRL Web **http://www.arrl.org/qexfiles/**. Look for 0303Youn.ZIP.
[29]M. Frerking, *Digital Signal Processing in Communication Systems* (Boston, Massachusetts: Kluwer Academic Publishers, ISBN 0-442-01616-6), pp 305-391.
[30]J. Stephenson, KD6OZH, "The ATR-2000: A Homemade High-Performance HF Transceiver, Pt 1," *QEX,* Mar/Apr 2000, pp 3-4.

Introducing a NEW CD-ROM series
on Antennas and Propagation for Wireless Communications

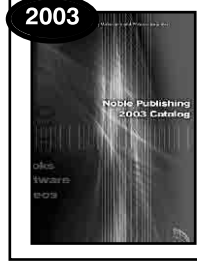
Antenna Fundamentals
Steven R. Best, Ph. D.

Antenna Fundamentals introduces basic antenna concepts and definitions used in the antenna industry. Antenna characteristics such as VSWR, radiation patterns, directivity, gain, polarization, axial ratio, and EIRP are defined and their impact on wireless system performance is discussed. The course also describes different antenna types, including resonant antennas, frequency independent antennas, aperture antennas, arrays and electrically small antennas.

CONTENT
Section 1: **Fundamental Antenna Properties**
antenna definition • electromagnetic waves • transmission waves
Section 2: **Antenna Performance Characteristics**
impedance • VSWR • bandwidth • radiation patterns • directivity
gain • polarization
Section 3: **Classification of Antennas**
antenna types • size • directivity

2003, CD-ROM ISBN 1-884932-36-3
NP-53 . . . . . $89.00

NOW AVAILABLE



View and shop from our latest printed catalog at

www.noblepub.com

Order on-line now!

NOBLE PUBLISHING
Noble Publishing Corporation
630 Pinnacle Court
Norcross, GA 30071
USA

4 WAYS TO ORDER
CALL 770-449-6774 · FAX 770-448-2839
E-MAIL orders@noblepub.com
INTERNET www.noblepub.com